Modular kMC in Fortran

LAMMPS Architecture in Fortran

2024/06/27, Nicolas Salles, CECAM-School

Modularity in “Software”

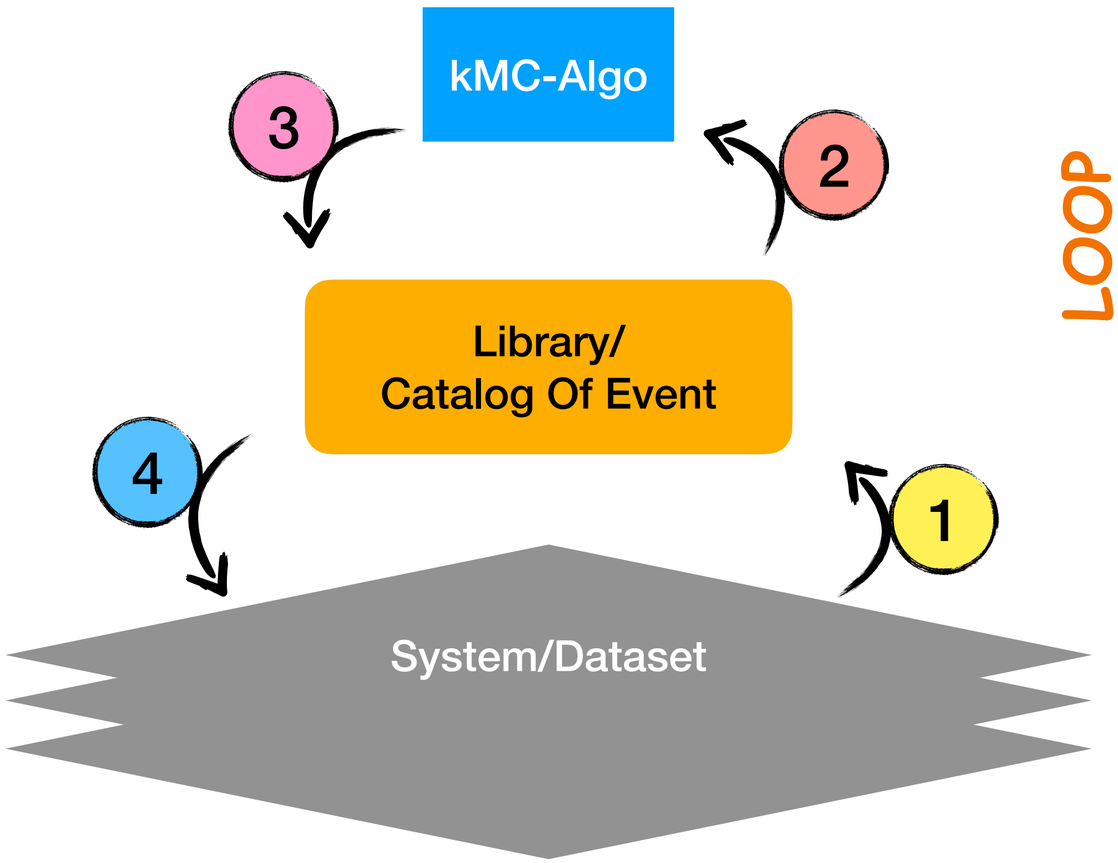
Some definition

Simulation: Initial data with specific WorkFlow

WorkFlow: Succession of task acting on data

Modularity: Be able to change each degree of freedom of the workflow

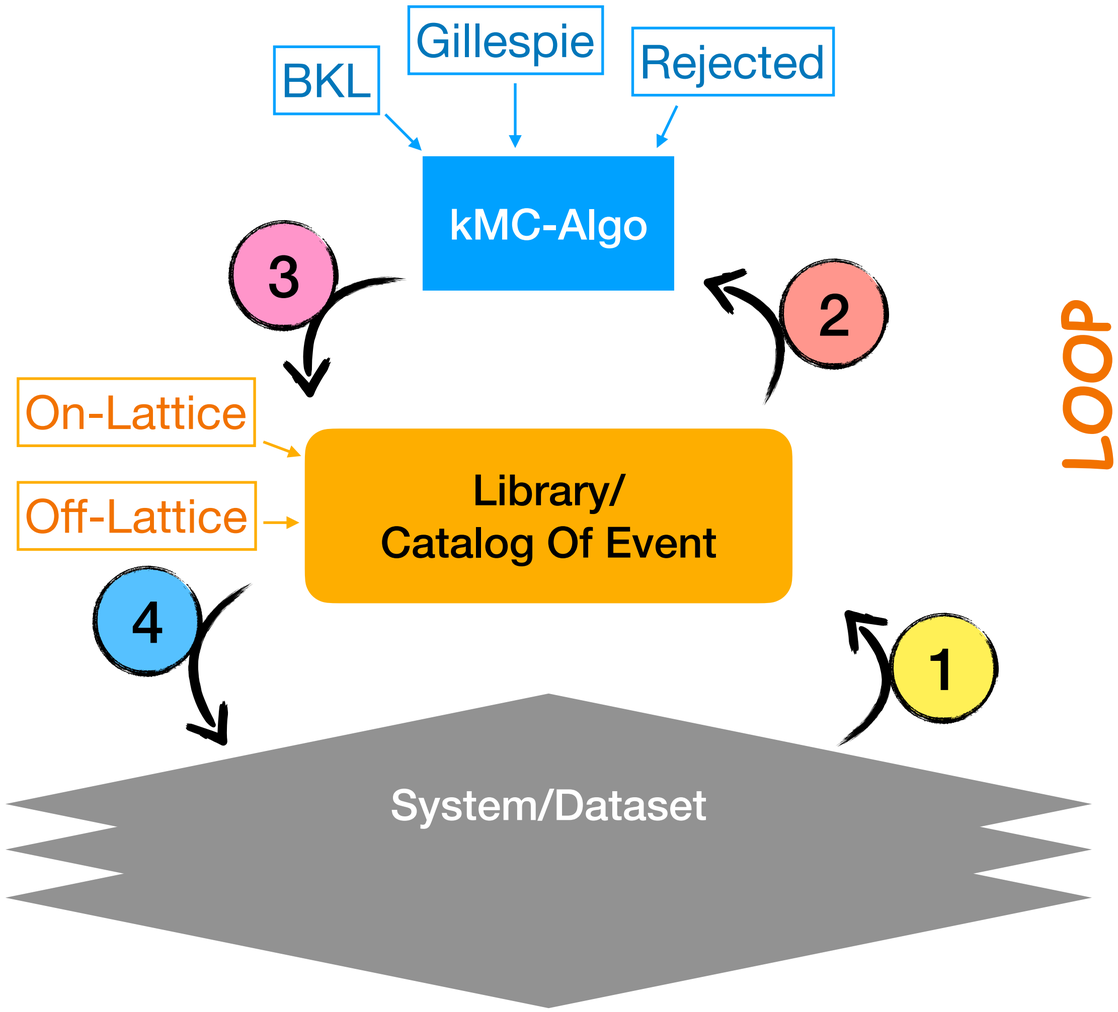
kMC WorkFlow



LOOP

1. Compare the system to the catalog of event
2. Give the list of possible event to the kMC algorithm
3. Return the site and the event selected
4. Apply the event on the selected site

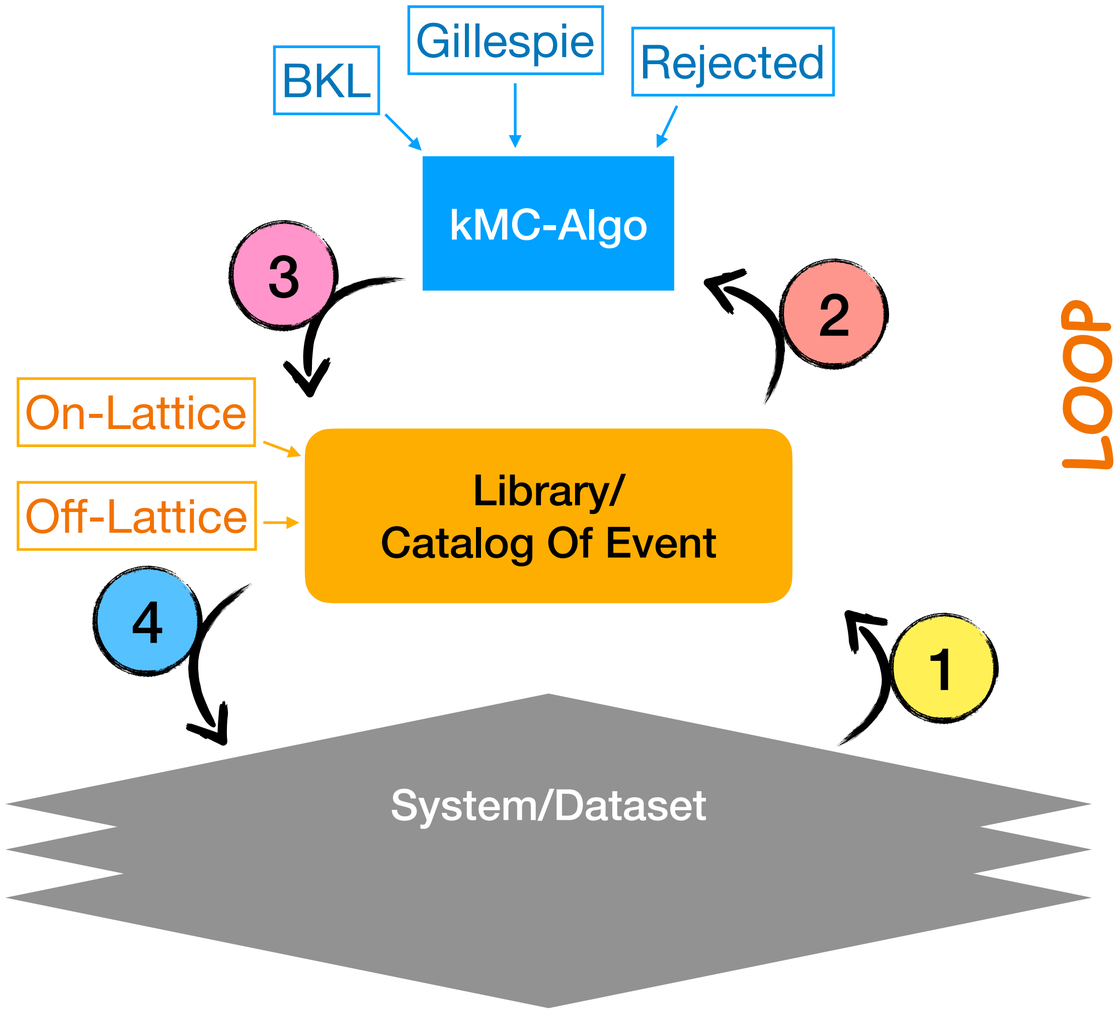
kMC WorkFlow



LOOP

1. Compare the system to the catalog of event
2. Give the list of possible event to the kMC algorithm
3. Return the site and the event selected
4. Apply the event on the selected site

kMC WorkFlow



LOOP

1. Library.Compare_system()
2. kMC.Select_event()
3. Library.Apply_event()

DATA.System

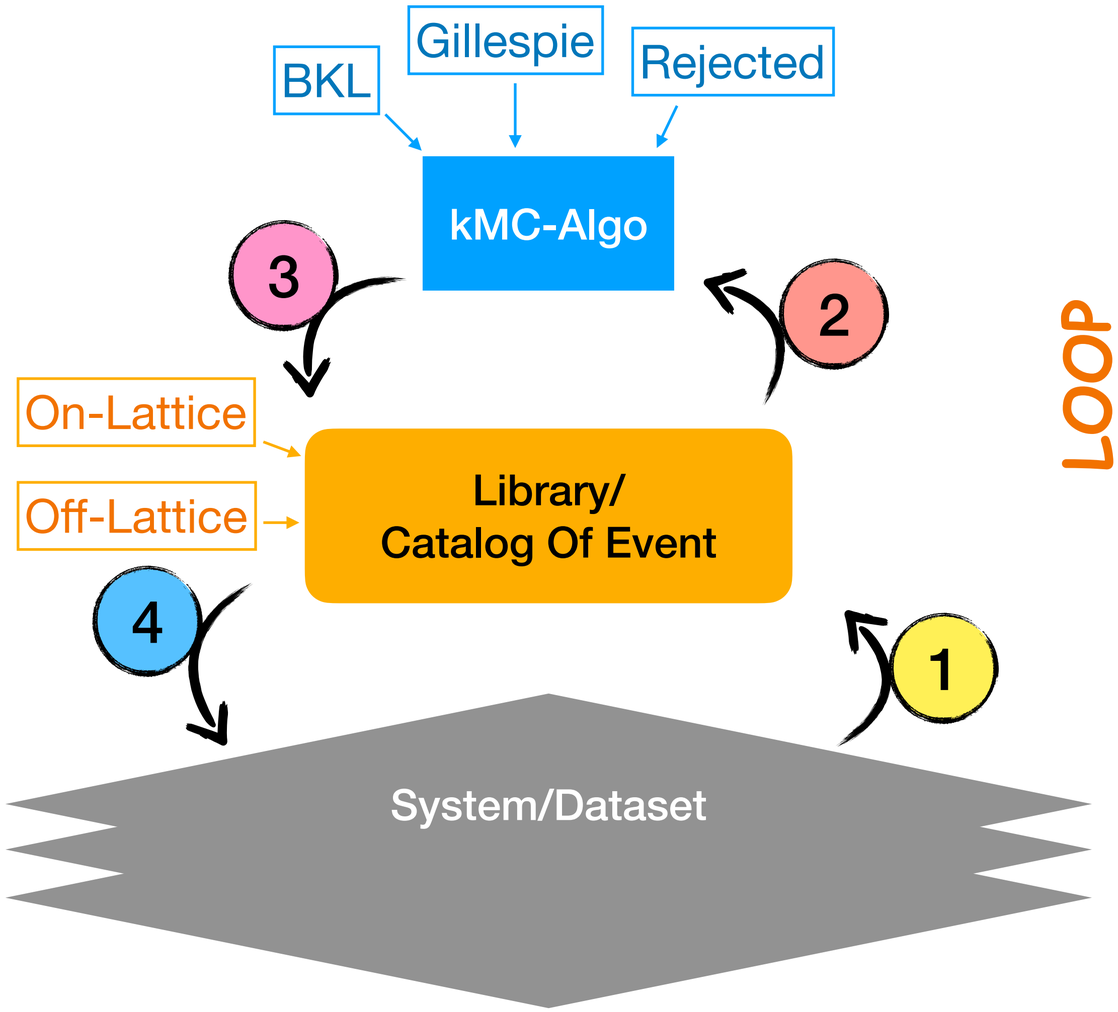
DATA.List_event
DATA.List_rate

DATA.Event
DATA.Site

DATA.System

Interaction between different kind of object and action

kMC WorkFlow



Give the opportunity to customise the simulation

LOOP

1. Library.Compare_system()
   Modify.Pre_Selection()
2. kMC.Select_event()
   Modify.Pre_Application()
3. Library.Apply_event()
   Modify.End_Step()

DATA.System

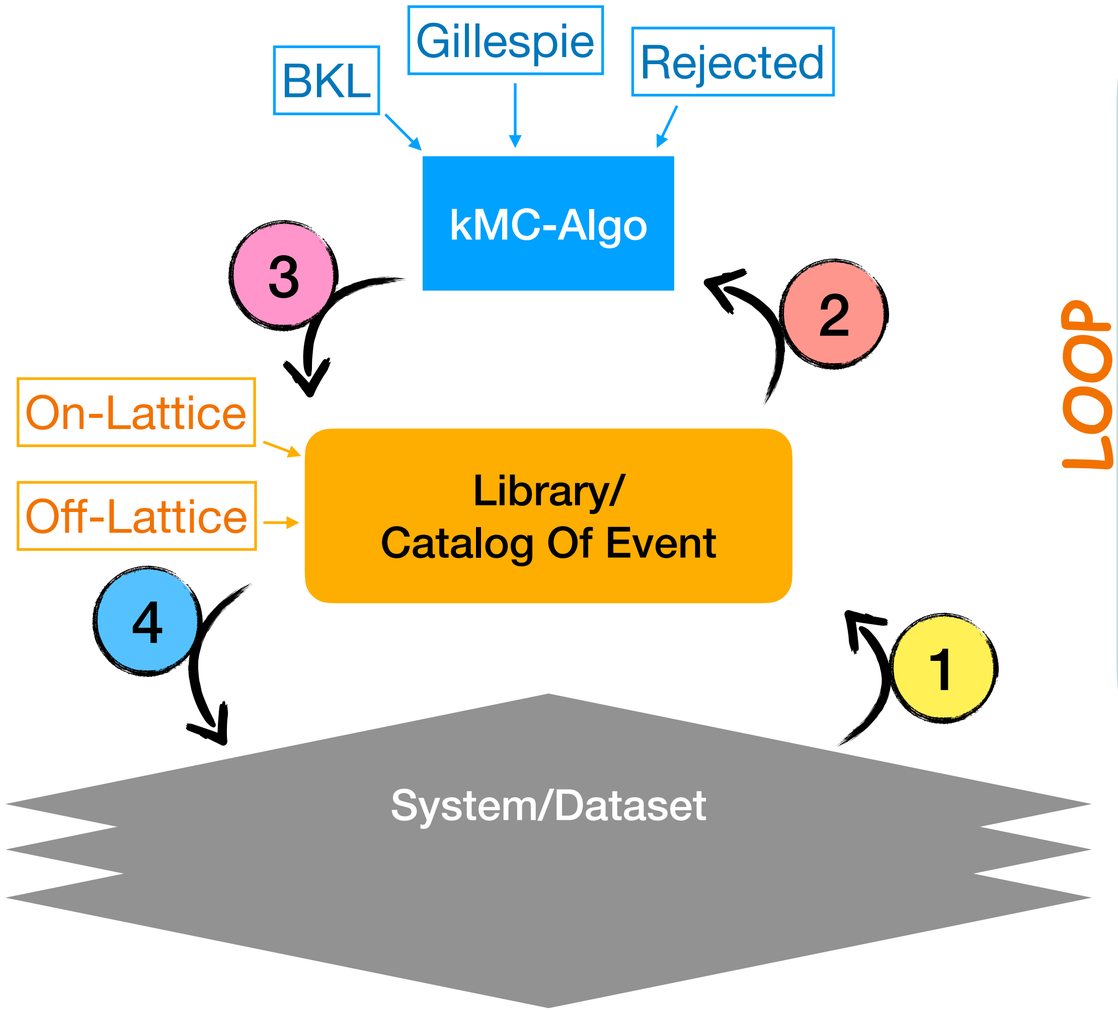
DATA.List_event
DATA.List_rate

DATA.Event
DATA.Site

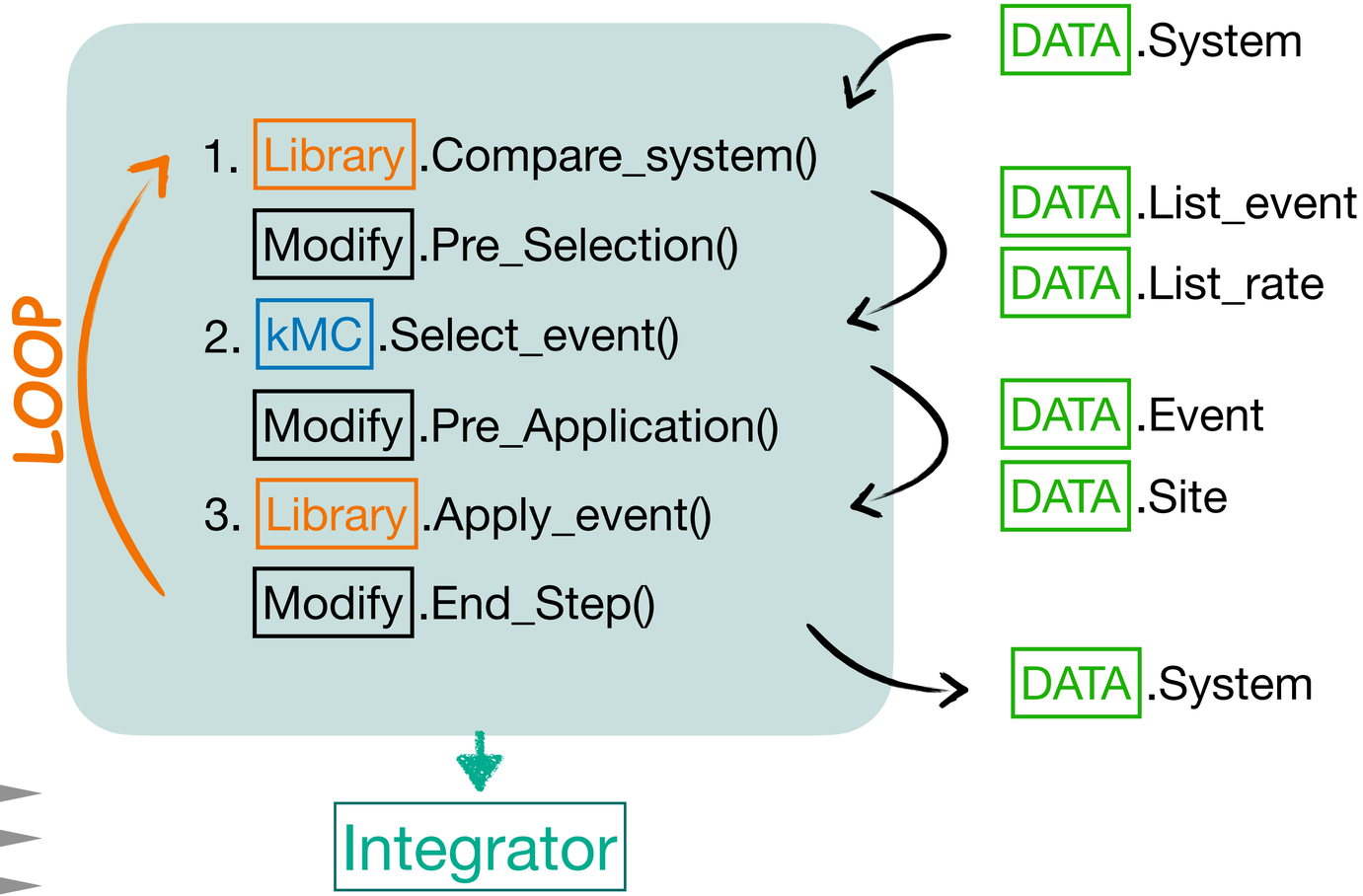
DATA.System

- Change the temperature, pressure, …
- Bias the rate of event
- Minimize the system if needed

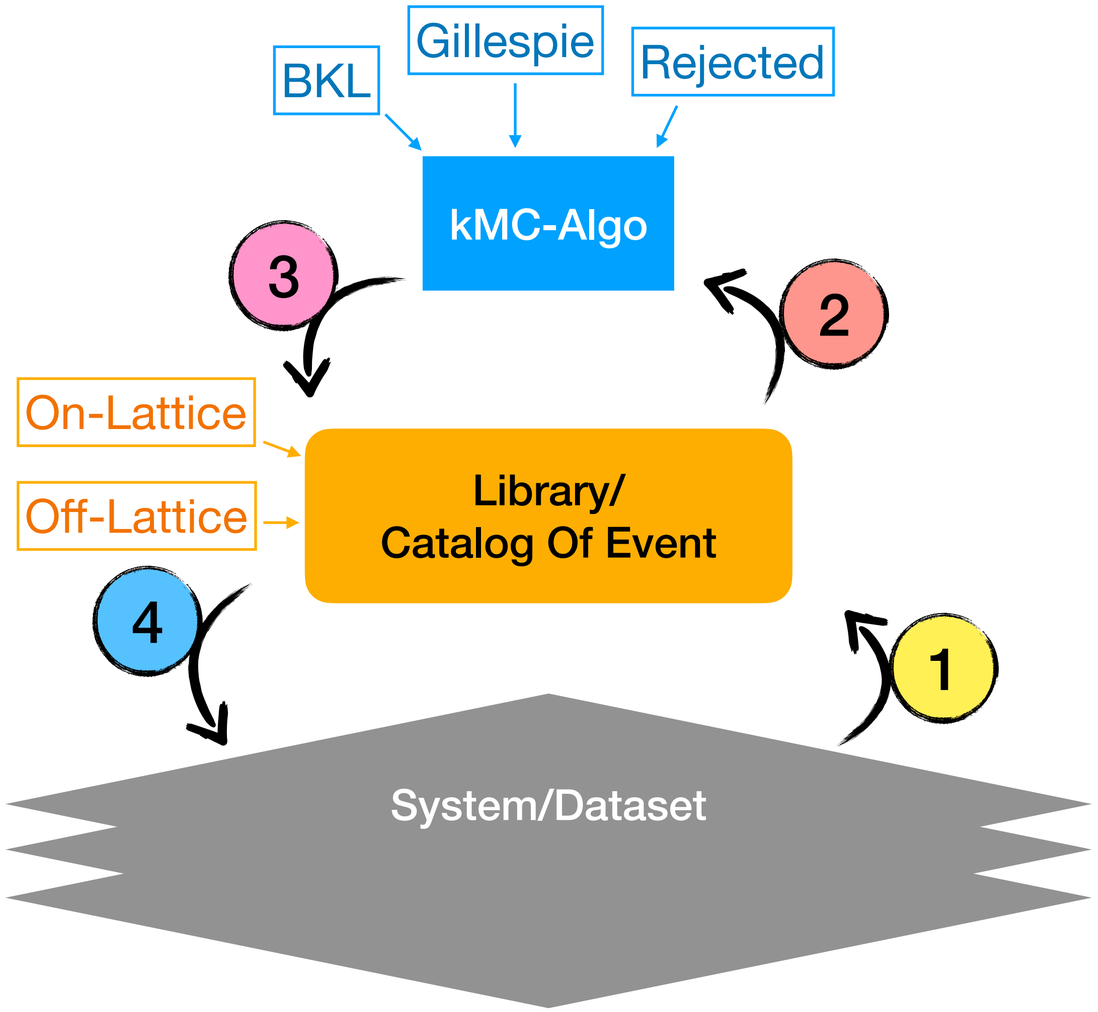
kMC WorkFlow



Encapsulate the time step kMC integration



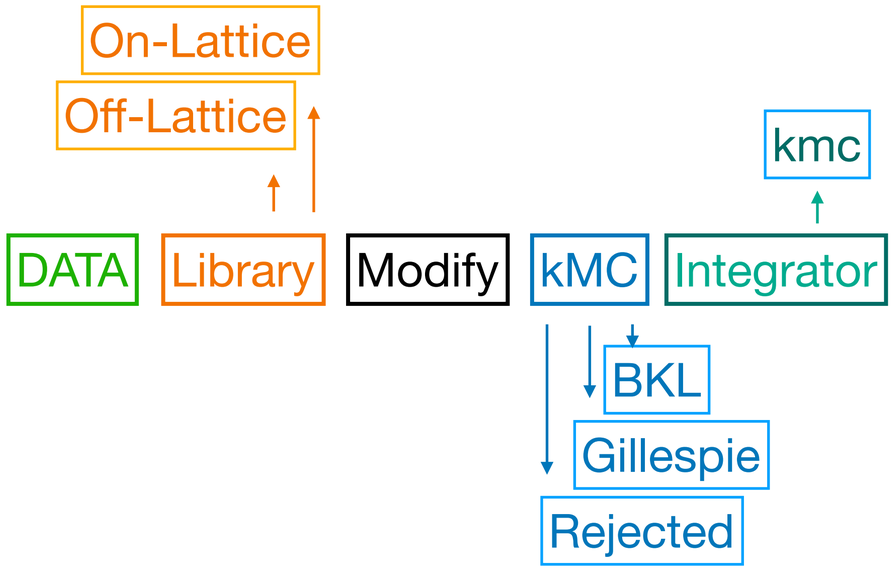
kMC WorkFlow

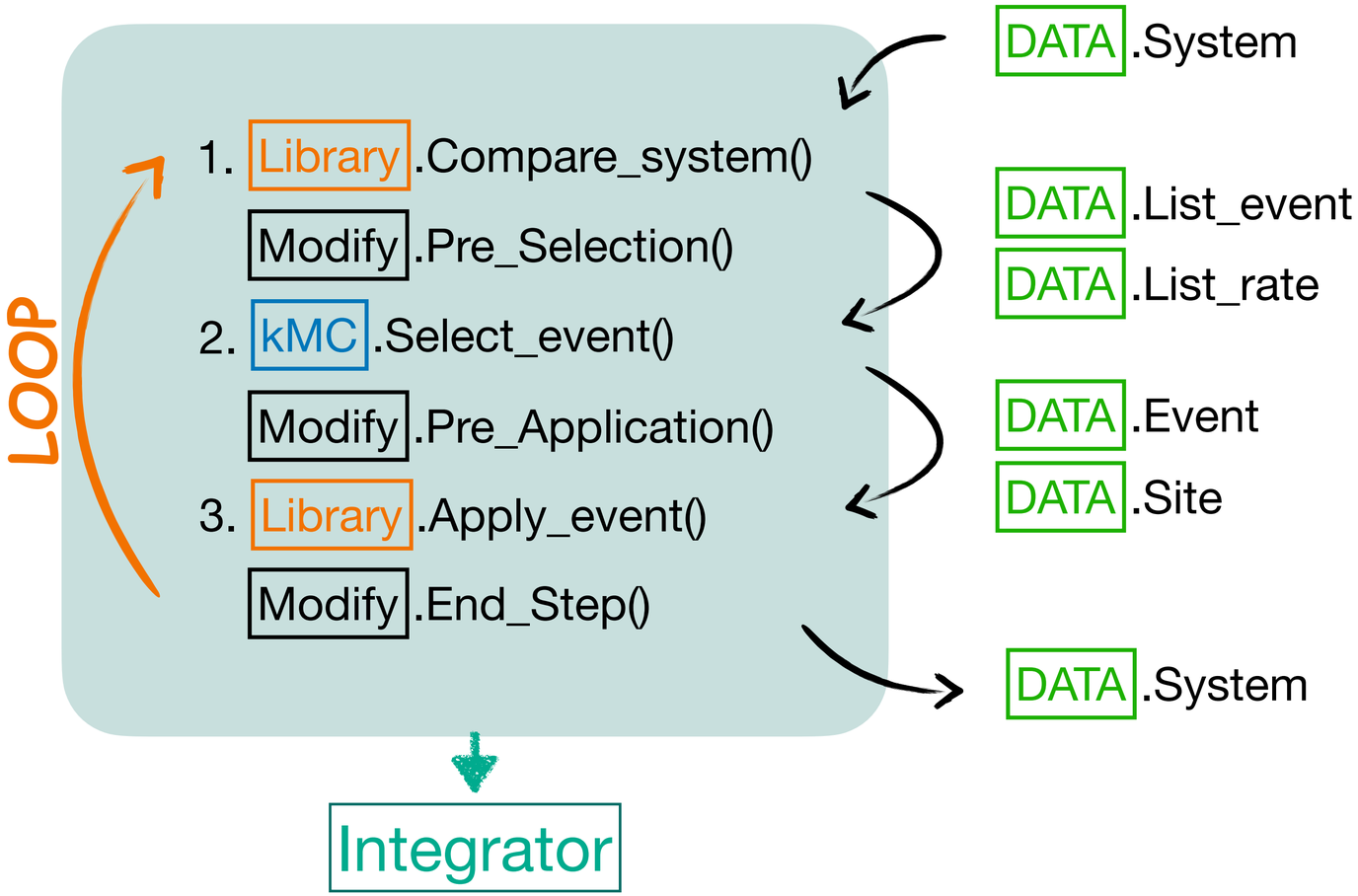



```
do i = 1,n  ! is better to start at 0

   ! ...Test on the maxtime
   if( update% check_maxtime() )then
     call bugs% warning( here, "Maxtime reached", &
                         [update% maxtime] )
     exit
   endif

   self% step = i

   call modify% start_step()     !! change temperature, ...

   !%! Check the neighbor list

   call force% library% system_state()  !! look at the event known i

   call modify% pre_integrate()         !! change temperature, ...

   call modify% initial_integrate()     !! Compute rate, select event

   call force% library% apply_event()   !! Apply the selected event

   call modify% post_integrate()        !! pre_minimize

   call modify% final_integrate()

   call modify% end_step()              !! after fix_minimize

   call output% writing( i )

enddo
```

Integrate has to access to classes modify, force, output, …

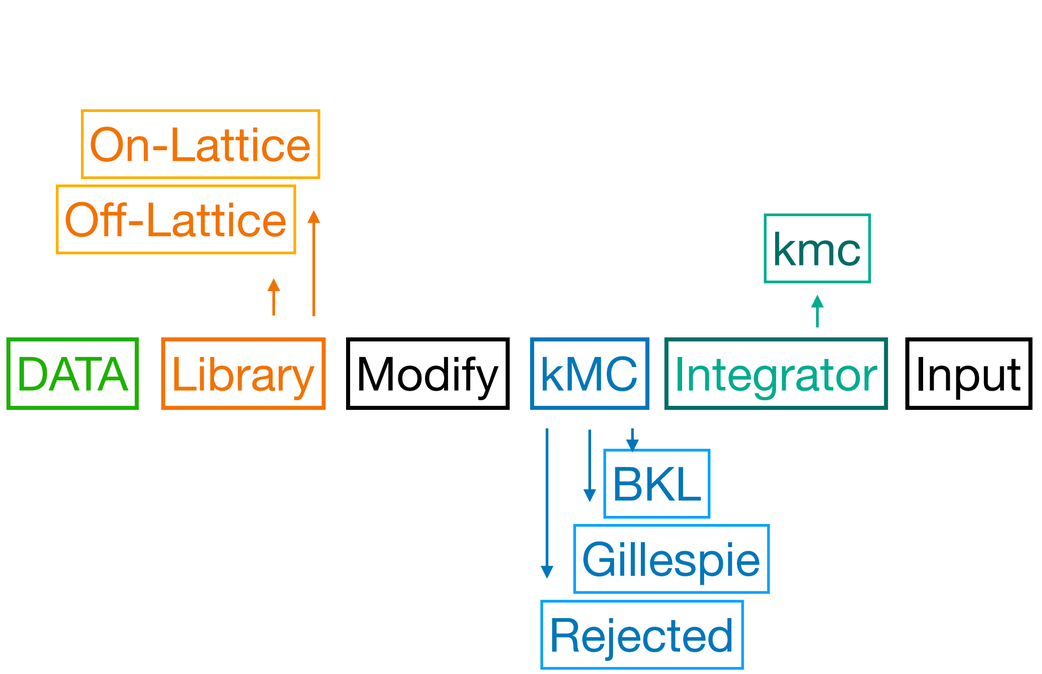
Software Organisation

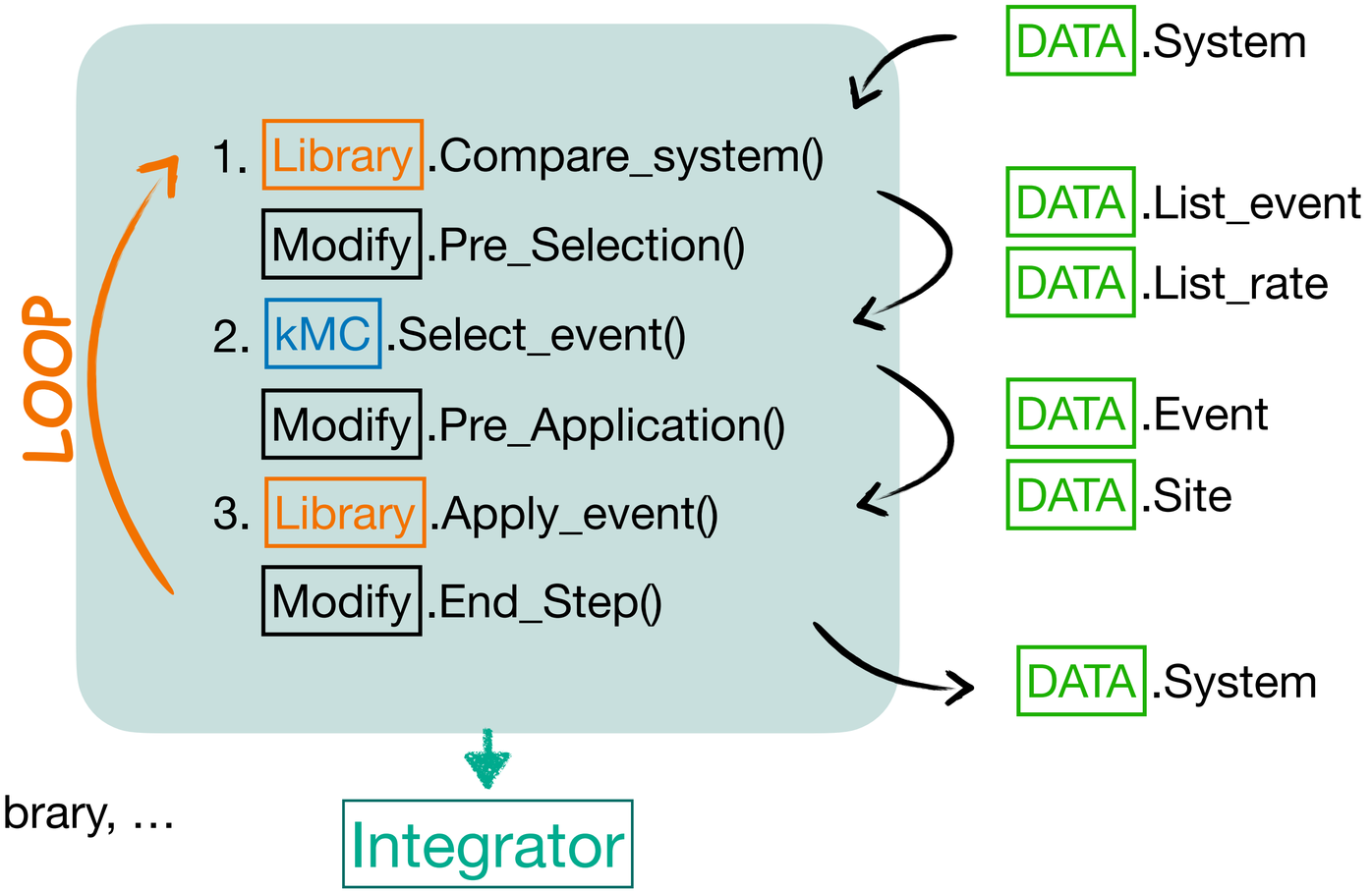




Through the integrator, the workflow is define and thanks to the polymorphism of class can customise its “colour”

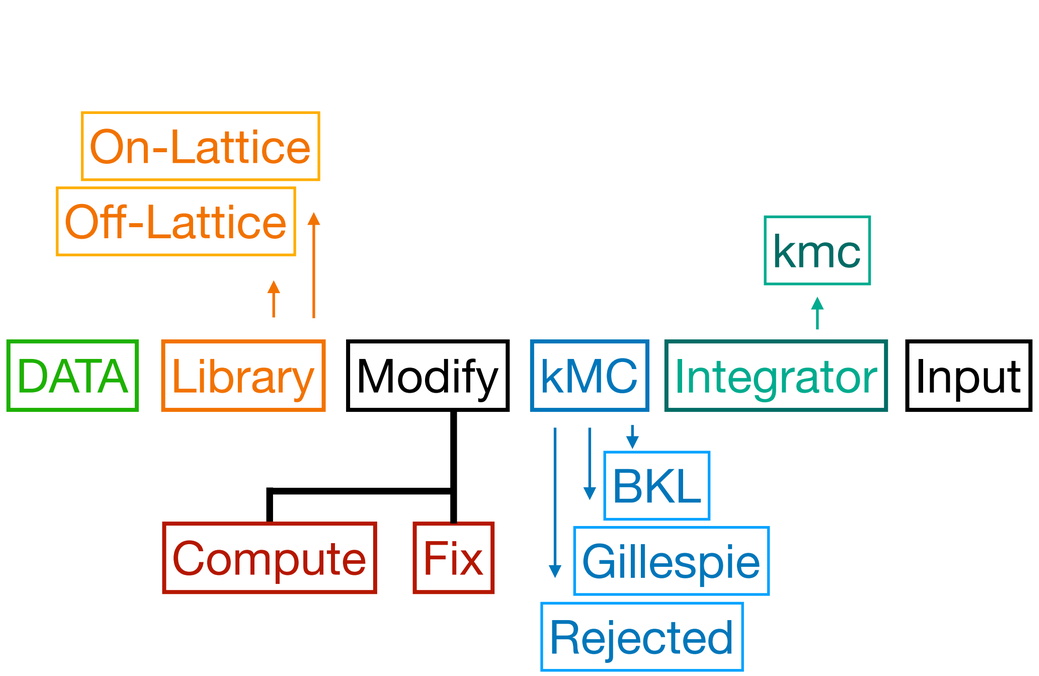
Software Organisation

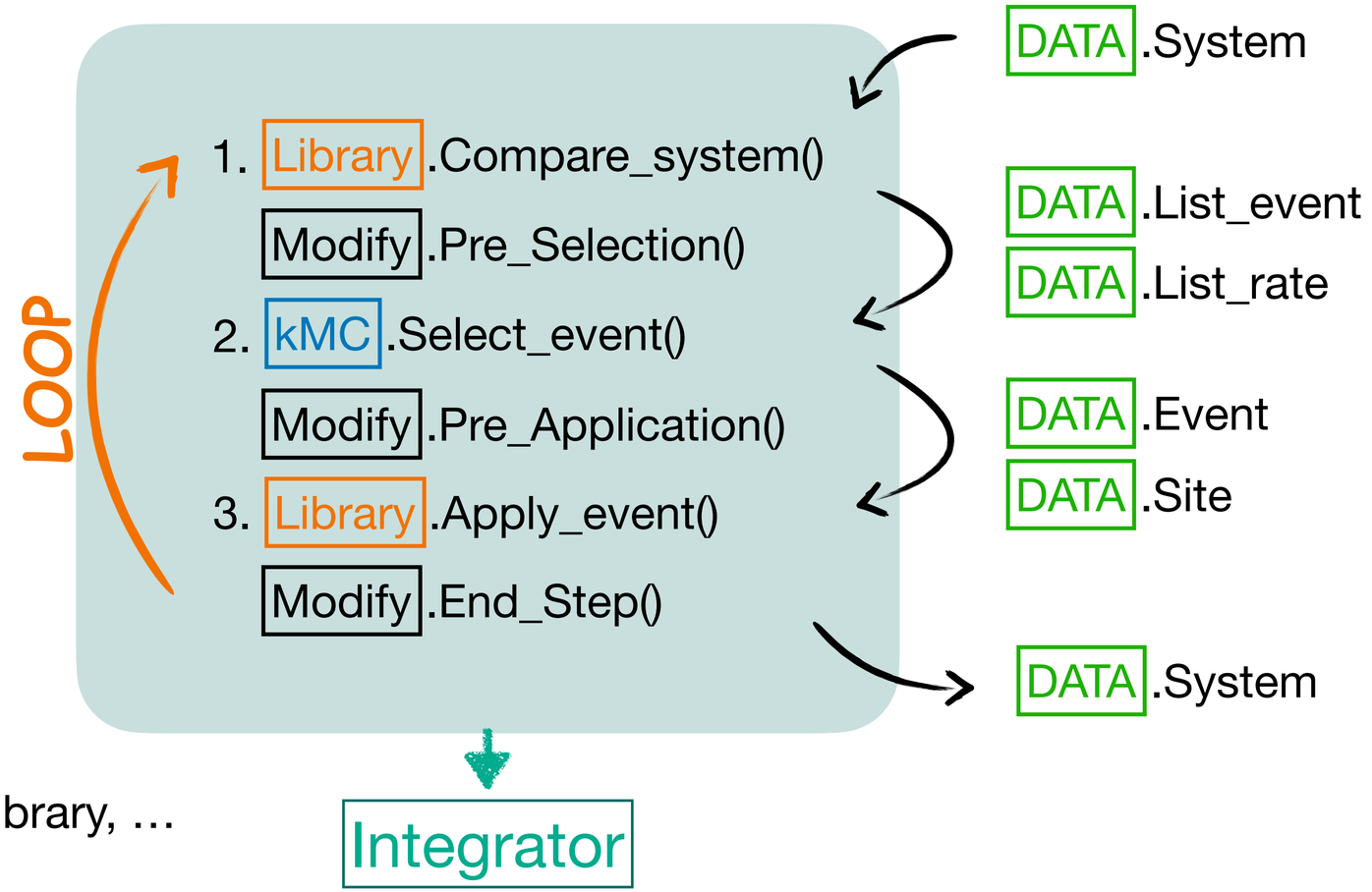




Input class manage the selection of various kMC, Library, …

Software Organisation



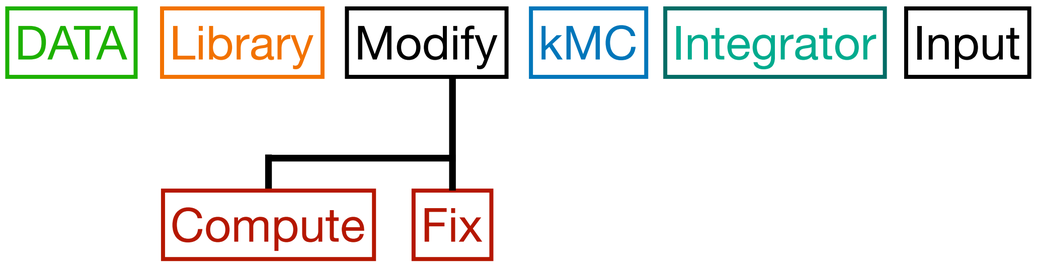


Input class manage the selection of various kMC, Library, …

Software Organisation

Horizontal Connection

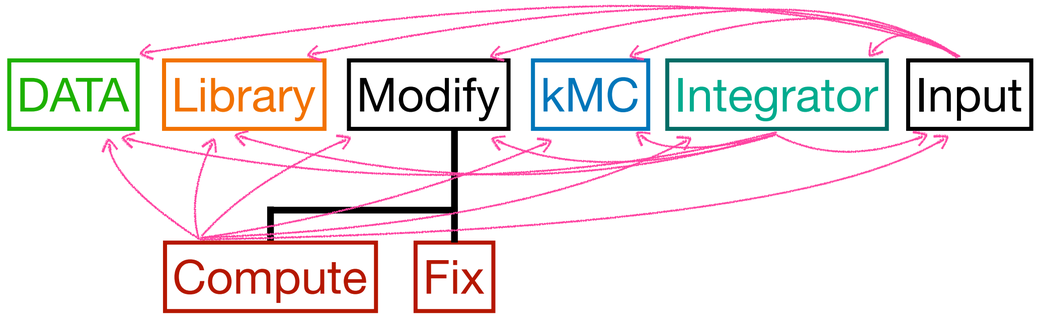
Guarantee the compartmentalization of each class:



Software Organisation

Horizontal Connection

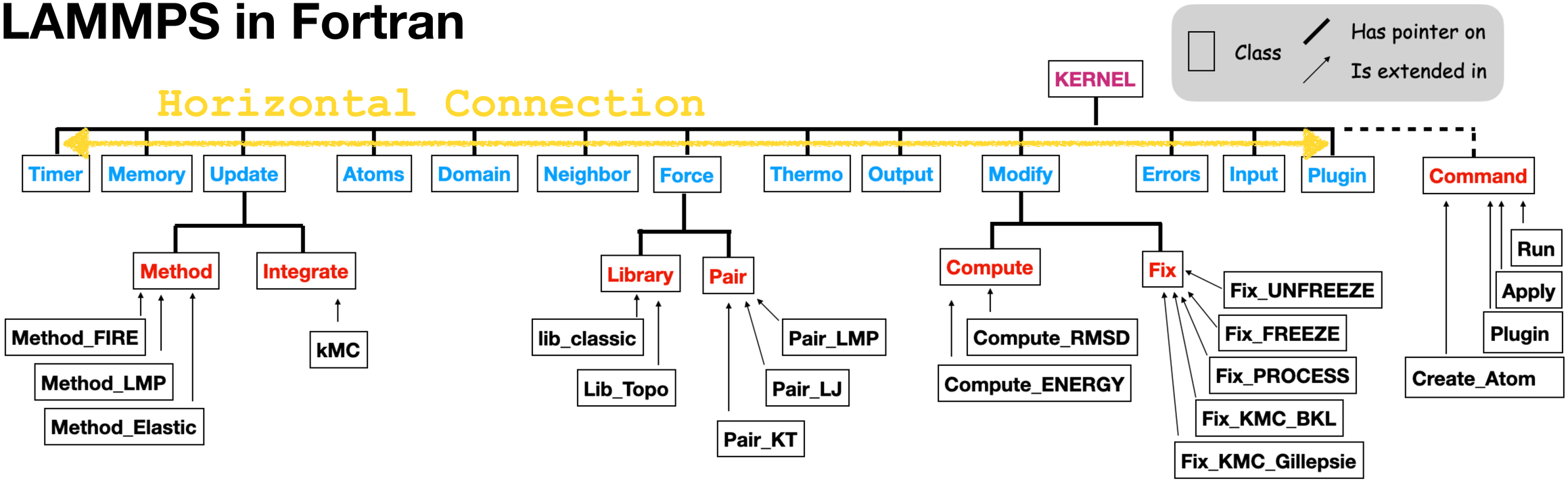
Guarantee the compartmentalization of each class: **Multiple linked-list**



Each class has a list of pointers to each of the other classes

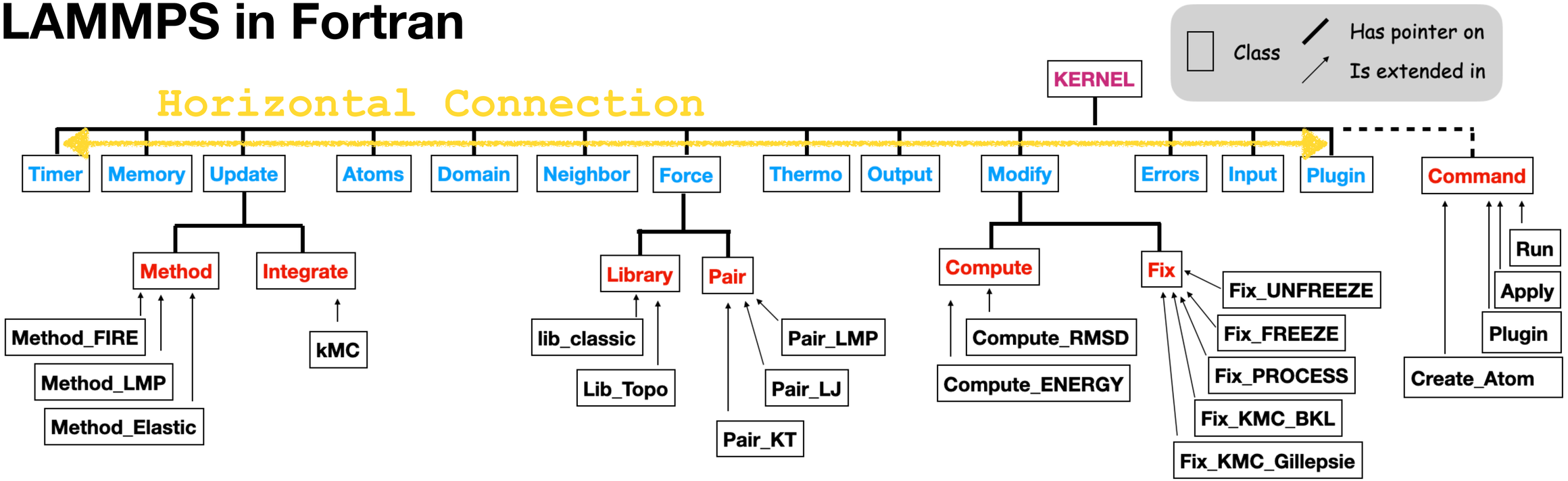
FLAMMPS

LAMMPS in Fortran



FLAMMPS

LAMMPS in Fortran

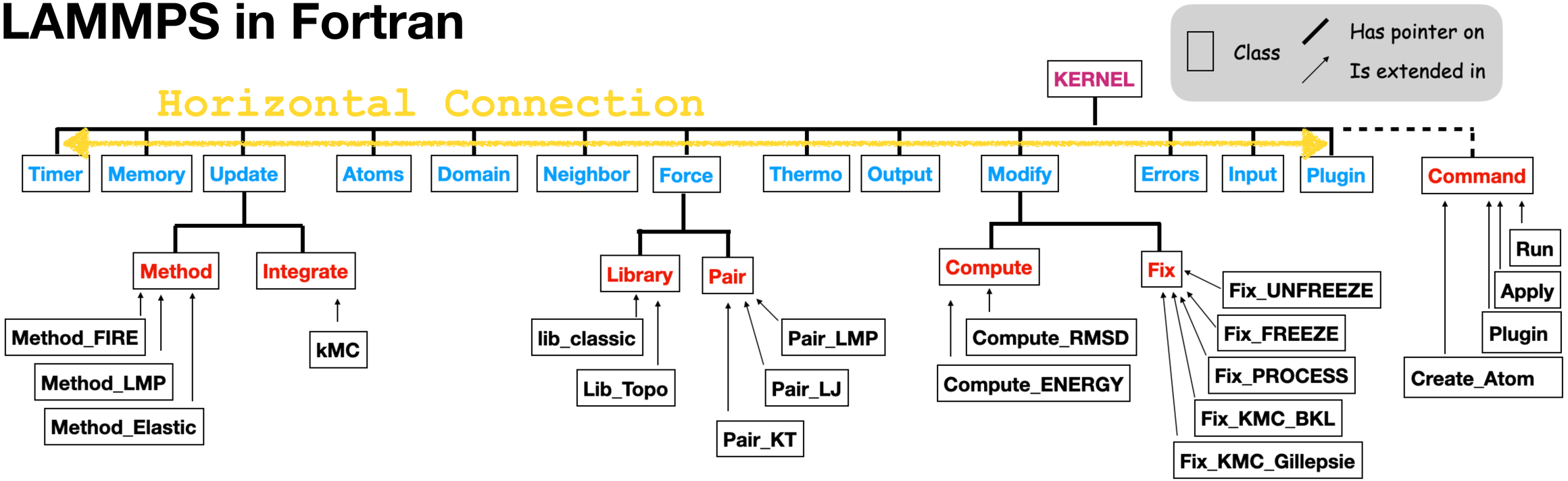


API:

```
Module kernel_api
 Contains
  type( c_ptr )function kernel_create()result( dkernel_cptr )bind(C, name="kernel_create")
  subroutine kernel_free( dk_cptr ) bind(C, name="kernel_free")
  subroutine kernel_command( dk_cptr, ccmd, ierr )bind(C, name="kernel_command")
  module function c2f_char( cstring )result(fstring)
```

FLAMMPS

LAMMPS in Fortran



Advantage: Interoperability

Interface only on the Class kernel

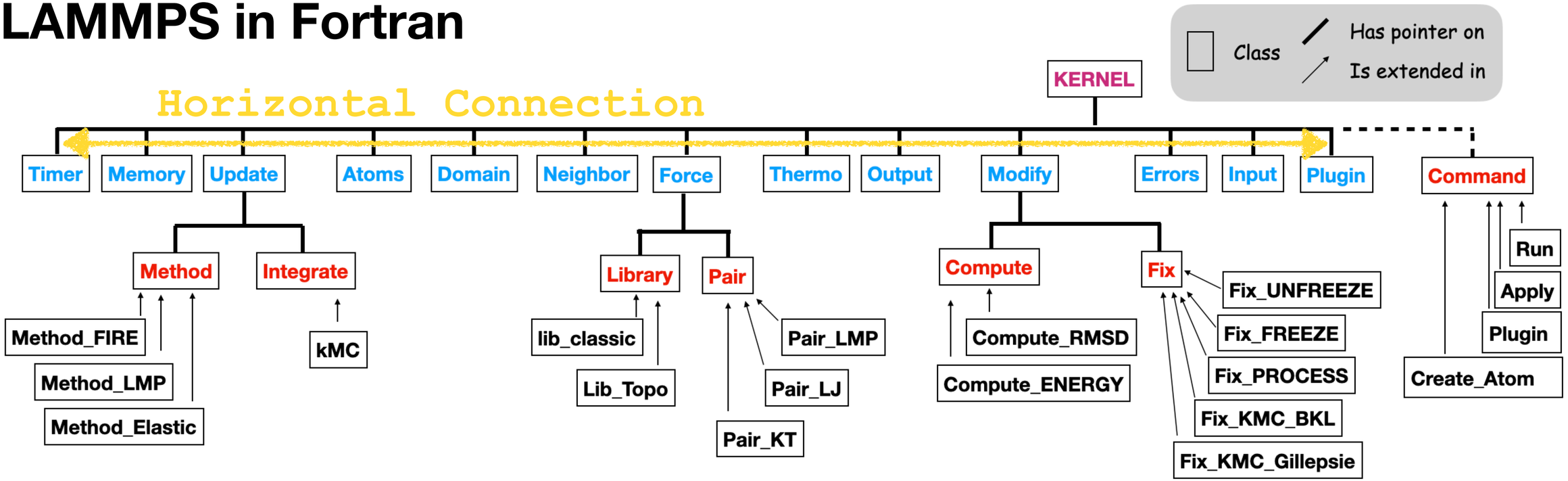
```
>>> import kernelpy as Kpy
>>> sim = Kpy.kernel()
```

Access to whole architecture thank to few routine

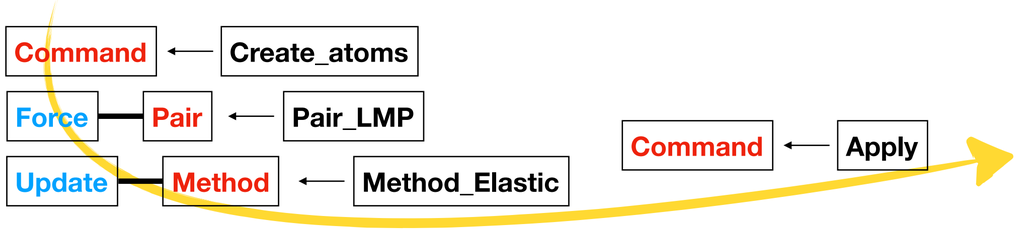
```
>>> sim.command("library_style classic")
```

FLAMMPS

LAMMPS in Fortran



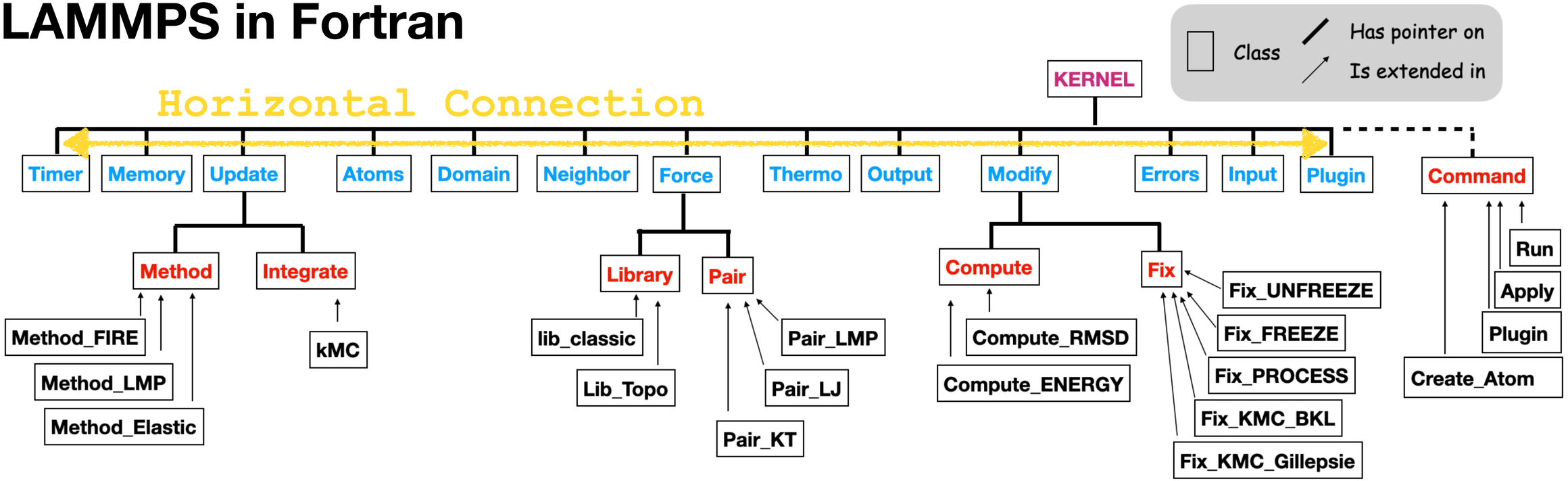
Advantage: WorkFlow Control



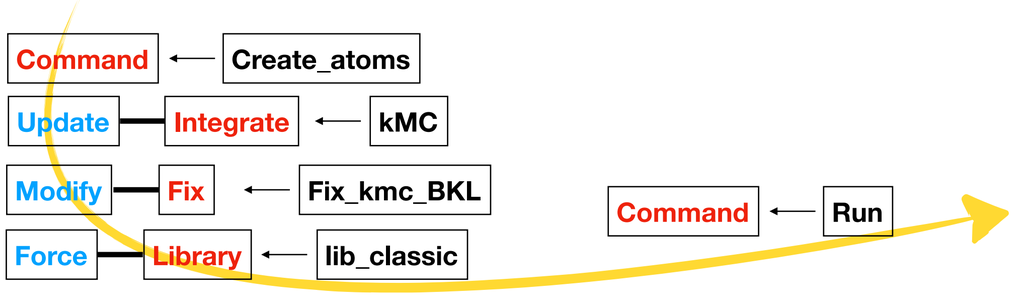
Compute
Elastic Constante
(LAMMPS Pair potential)

FLAMMPS

LAMMPS in Fortran



Advantage: WorkFlow Control



Run kMC simulation
with on lattice library

How to…

The Modularity of LAMMPS is not only due to the C++ language

OO Prgramming → Horizontal connection

BASH script associated to preprocessing MACRO → the automatic incorporation of new derived classes at compile time

How to…

Fortran specificity

Some essential tools for the OOP in Fortran

- **Derived type**: equivalent to the `class` in C++
- **Module**: equivalent to the header file (`.h`) in C++
- **Submodule**: equivalent to the file `.cpp` in C++

Some rules in Fortran

- `header modules` contains the declaration of all the **elements** of the type
  - → Have to be compiled before their use

- `Submodule` contains the functions and subroutines declared in the header
  - → are compiled after the compilation of all header modules
  - → Dependencies to other `headers` are declared (use) in `submodule`

How to…

Fortran specificity

Some essential tools for the OOP in Fortran

- **Derived type**: equivalent to the `class` in C++
- **Module**: equivalent to the header file (`.h`) in C++
- **Submodule**: equivalent to the file `.cpp` in C++

```
Module atom_h
  Type t_atom
    Integer :: natom
    …
   Contains
    Procedure :: routines, …
  End type

  Interface t_atom
    Module procedure :: atom_constructor
  End interface

  Interface
    Module subroutine <prototype>
    Module function <prototype>
    …
  End interface
End Module atom_h
```

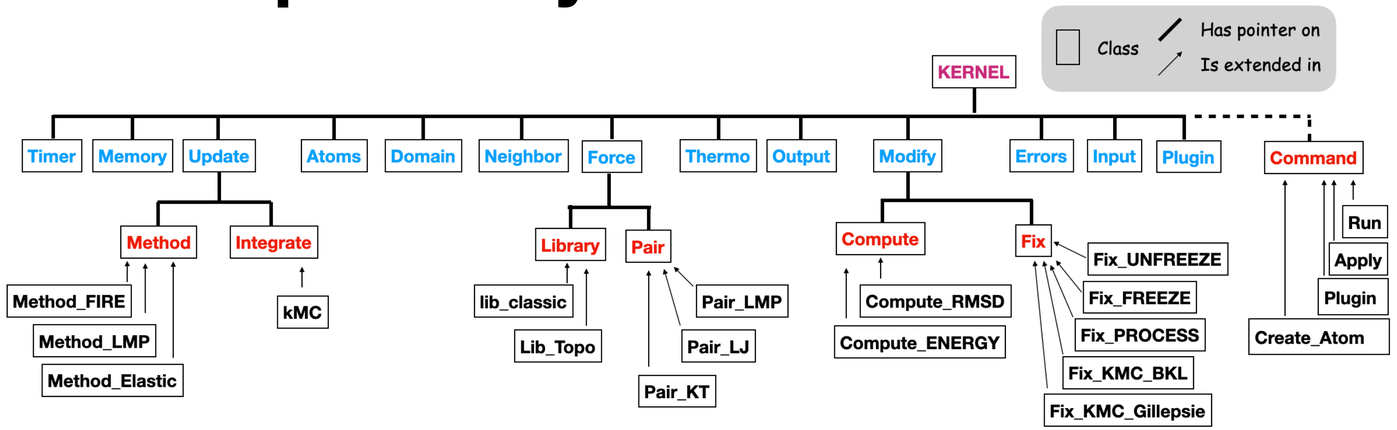
Some rules in Fortran

- `header modules` contains the declaration of all the **elements** of the type
  - → Have to be compiled before their use
- `Submodule` contains the functions and subroutines declared in the header
  - → are compiled after the compilation of all header modules
  - → Dependencies to other `headers` are declared (use) in `submodule`

```
submodule( header_atom ) cpp_atom
  Use other_header
 contains
  module subroutine <defintion>
  module function <definition>
End submodule cpp_atom
```

How to…

Fortran specificity




```
Type :: t_kernel
  type( t_atom ),     pointer :: atom
  type( t_domain ),   pointer :: domain
  type( t_error ),    pointer :: error
  type( t_force ),    pointer :: force
  type( t_input ),    pointer :: input
  type( t_modify ),   pointer :: modify
  type( t_neighbor ), pointer :: neighbor
  type( t_output ),   pointer :: output
  type( t_thermo ),   pointer :: thermo
  type( t_update ),   pointer :: update
End type t_kernel
```

How to…

Twin Pointers Class

```
Type :: t_kernel
  type( t_atom ),     pointer :: atom
  type( t_domain ),   pointer :: domain
  type( t_error ),    pointer :: error
  type( t_force ),    pointer :: force
  type( t_input ),    pointer :: input
  type( t_modify ),   pointer :: modify
  type( t_neighbor ), pointer :: neighbor
  type( t_output ),   pointer :: output
  type( t_thermo ),   pointer :: thermo
  type( t_update ),   pointer :: update
End type t_kernel
```

The twin pointer of kernel

```
type :: pointers
  class(*), pointer :: atom
  class(*), pointer :: domain
  class(*), pointer :: error
  class(*), pointer :: force
  class(*), pointer :: input
  class(*), pointer :: modify
  class(*), pointer :: neighbor
  class(*), pointer :: output
  class(*), pointer :: thermo
  class(*), pointer :: update
  class(*), pointer :: kernel
  […]
end type pointers
```

Plus the kernel

```
Module atom_h
  […]
  type, extend( pointers ) :: t_atom
    […]
  end type t_atom
  Interface t_atom
    Procedure :: atom_constructor
  End interface
  Interface
    module function atom_constructor()result( this )
      type( atom ), pointer :: this
    end function atom_constructor
  End interface
end module atom_h
```

Each Class is an extension of type `pointers` and is “multiple-linked” at constructor time

How to…

Access to the other class

Now all classes has `class(*), pointer` on each **`main class`**

To use these polymorph pointers, Fortran imposes to identify them

- Identification Procedure in **submodule**:

```
module subroutine something_on_atom( self )
  Use error_h, only : t_error
  class( t_atom ), intent( inout ) :: self

  type( t_error ), pointer :: err

  select type( self% errors )
    type is( error ); err => self% errors
  end select

  Call err% error(_FILE_, _LINE_,“There is a problem!!”)

end subroutine something_on_atom
```

```
type :: pointers
  class(*), pointer :: atom
  class(*), pointer :: domain
  class(*), pointer :: errors
  class(*), pointer :: force
  class(*), pointer :: input
  class(*), pointer :: modify
  class(*), pointer :: neighbors
  class(*), pointer :: output
  class(*), pointer :: thermo
  class(*), pointer :: update

  class(*), pointer :: kernel
  […]
end type pointers
```

How to…

Access to the other class

Now all classes has `class(*), pointer` on each **`main class`**

To use these polymorph pointers, Fortran imposes to identify them

- Identification Procedure in **submodule**:

```
module subroutine something_on_atom( self )
  Use error_h, only : t_error
  class( t_atom ), intent( inout ) :: self

  type( t_error ), pointer :: err

  select type( self% errors )
    type is( error ); err => self% errors
  end select

  Call err% error(_FILE_, _LINE_,“There is a problem!!”)

end subroutine something_on_atom
```

Each time you want to access to another `main class`

```
type :: pointers
  class(*), pointer :: atom
  class(*), pointer :: domain
  class(*), pointer :: errors
  class(*), pointer :: force
  class(*), pointer :: input
  class(*), pointer :: modify
  class(*), pointer :: neighbors
  class(*), pointer :: output
  class(*), pointer :: thermo
  class(*), pointer :: update

  class(*), pointer :: kernel
  […]
end type pointers
```

How to…

Access to the other class

Define an identification procedure for each **`main class`**

```
module subroutine link_error( this, selector )
  use pointers_h
  class( pointers ),     intent( in ) :: this
  type( error ), pointer, intent( inout ) :: selector
  select type( this% errors )
    type is( error ); selector => this% errors
  end select
end subroutine link_error
```

→ Define generic name

```
interface link_class
  module procedure :: link_atom
  module procedure :: link_input
  module procedure :: link_error
  […]
end interface
```

```
module subroutine something_on_atom( self )
  Use error_h, only : t_error
  Use kernel_k, only : link_class
  class( t_atom ), intent( inout ) :: self

  type( t_error ), pointer :: err

  select type( self% errors )
    type is( error ); err => self% errors
  end select
  call link_class( self, err )

  call err% error(_FILE_, _LINE_,“There is a problem!!”)

end subroutine something_on_atom
```

How to…

Synthesis

Horizontal connection in Fortran

```
class(*), pointer ::
select type()
call link_class()
```

Repeated for each class

How to…

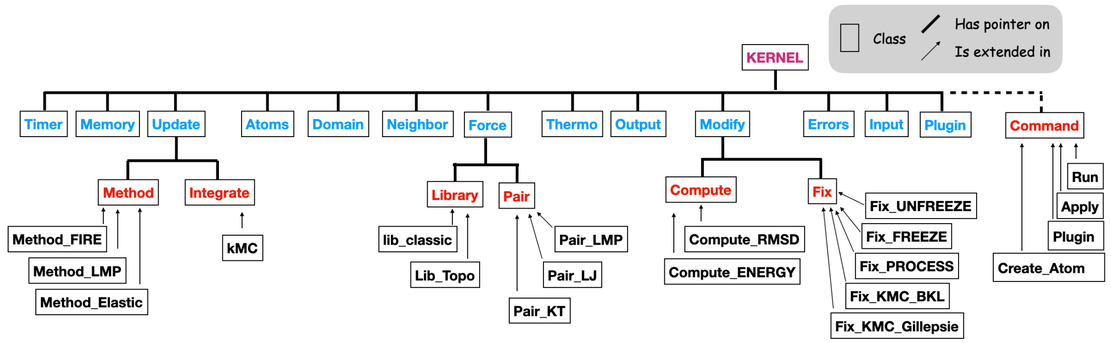
Synthesis

Horizontal connection in Fortran

```
class(*), pointer ::
select type()
call link_class()
```

Repeated for each class

→ Preprocessing MACRO to reduce the architecture



→

ROOT/

- **Pointers**
- **KERNEL**
- **Error**
- **Input**
- **Command**

Included at compile time

- Timer
- Memory
- Atom
- Domain
- Neighbor
- Thermo
- Modify
- Plugin
- Force

Interoperability based on kernel class → **Standardize the interfaces**

How to…

Synthesis

Horizontal connection in Fortran

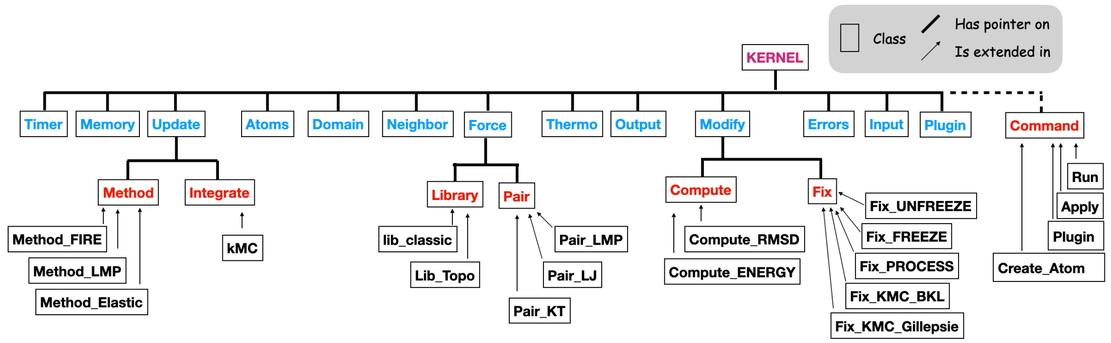
```
class(*), pointer ::
select type()
call link_class()
```

Repeated for each class

GITLAB Repo with tutorial soon…

https://gitlab.com/mammasmias/

→ Preprocessing MACRO to reduce the architecture



ROOT/

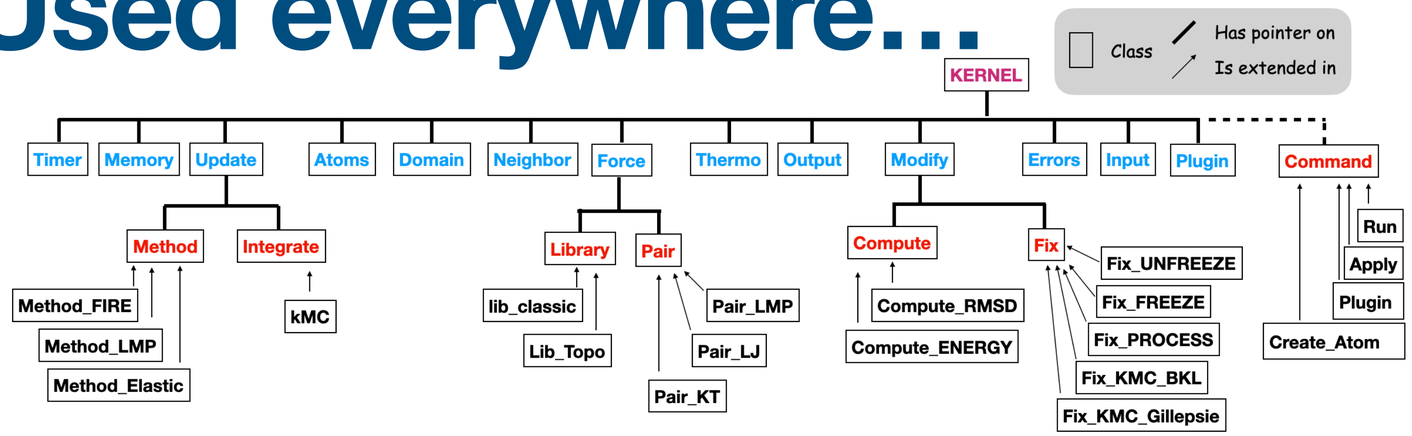
- Pointers
- KERNEL
- Error
- Input
- Command

Included at compile time

- Timer
- Memory
- Atom
- Domain
- Neighbor
- Thermo
- Modify
- Plugin
- Force

Interoperability based on kernel class → **Standardize the interfaces**

Used everywhere…




```
module kIRA_h
  […]
  type, extend( pointers ) :: t_kIRA
    type( t_gencat ), pointer :: gencat
    […]
  end type t_kIRA
  Interface t_kIRA
    Procedure :: kIRA_constructor
  End interface
  Interface
    module function kIRA_constructor()result( this )
      type( t_kIRA ), pointer :: this
    end function kIRA_constructor
  End interface
end module kIRA_h
```