

# Introduction



# Prerequisites

For this training session:

- Trio\_U (User's training session)
- C++ (Intermediate)

Later, if you want to develop/contribute to Trio\_U:

- Git (Basic)
- MPI (Basic)
- French skills (Mandatory)



# Objectives

To get a general knowledge of the Trio\_U code

To be able to look for useful information in the code for a specific development

To acquire reflexes to develop while following Trio\_U rules of coding



# Useful links

Trio\_U:

[ftp://ftp.cea.fr/pub/Trio\\_U/a87pour/index.html](ftp://ftp.cea.fr/pub/Trio_U/a87pour/index.html)

<mailto:triu@cea.fr>

C++:

<http://www.tutorialspoint.com/cplusplus>

Git:

<http://www-cs-students.stanford.edu/~blynn/gitmagic/index.html>

# Exercise: Trio\_U download

```
# If you are on a CEA Saclay PC:  
$ cd /export/home/yourlogin  
# From everywhere on the CEA network, download (~5mn) with:  
$ git clone git://is223196/official/trio_u  
# Go into your workspace :  
$ cd trio_u  
$ ls  
# Create a new work branch :  
$ git checkout -b YourNewBranch
```

# Trio\_U

## An object oriented CFD code

# Interest of Trio\_U

- Implement and test your numerical or physical models
- Reuse existing validated data structures
- Run your models on very large meshes thanks to parallelism
- Consolidate your work
  - Developments are integrated, documented, ported, tested, maintained by Trio\_U support team

# Interest of Trio\_U

- Need an investment:
  - to acquire the knowledge of the data structure
  - because of lack of documentation or obsolete one
  - to avoid several pitfalls (from C++ or Trio\_U)



# What is Trio\_U CFD code ?

## It provides :

- 2 spatial discretizations (VDF, VEF)
- Several time schemes
  - Explicit forward Euler, backward Euler, Runge Kutta 2-3-4,...
- Several schemes according the discretization
  - Quick, Upwind, EF\_stab, Muscl,...
- Templates to create new Equation, Problem, Field,...
- Several efficient tools to solve linear systems through the PETSc library :
  - Solvers : CG, BiCGstab, GMRES, Cholesky
  - Preconditioners : SSOR, ILU, Jacobi, Boomeramg, ....
- Data structures and functions to quickly parallelize your developments

# Trio\_U

- What can Trio\_U can handle
  - Runs on every Linux box (32/64 bits)
  - Runs on all the CEA clusters
    - Has already run a LES on a  $300.10^6$  cells mesh with 4600 cores (curie on CCRT)

# Trio\_U

## Specifications/Choices explained

# Main specifications:

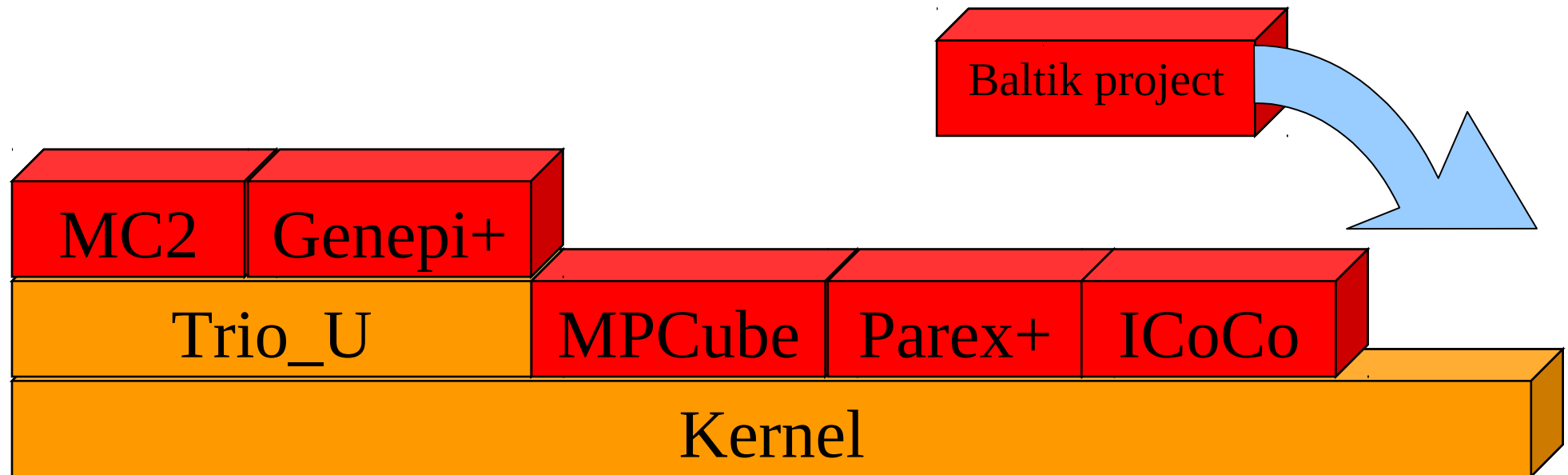
Enable developments with the following characteristics:

- fast
- reliable
- reusable
- effective
- documented
- enable encapsulation of Fortran modules

# Main Choices:

- **Object Oriented Conception** using UML method
  - Modularity, maintainability, library encapsulation
- **C++** implementation
  - Standard, performances, C/Fortran compatibility
- Parallelism by sending/receiving messages (**MPI**)
  - Standard, portable
- Multi-site configuration management (previously Clearcase, now **Git**)
  - Co-developing
- Automatic generation via Doxygen of **HTML** documentation from code sources
  - Documentation is up to date

## 2 ways to develop in Trio\_U



I) In Trio\_U/Kernel

II) In a Baltik project on top of Trio\_U/Kernel

In the 2 cases, you need to first install and build Trio\_U/Kernel.

Trio\_U 1.7.0 developer training session

# 2 ways to develop in Trio\_U

## I) Develop directly in Trio\_U/Kernel

- You want to contribute to Trio\_U/Kernel
- But if you want to share your work, you will need :
  - to follow the Trio\_U rules of coding
  - to check and respect the non regression of other parts of the code

## II) Develop in a Baltik project based on Trio\_U/Kernel

- You want to develop your own project
  - more freedom about rules of coding/non regression of the Trio\_U/Kernel
- Baltik means **B**uilding an **A**pplication **L**inked to **T**rio\_U **K**ernel

# Exercise: Trio\_U build

*# Basic build*

*\$ cd Trio\_U*

*# Configure (~5mn) :*

*\$ ./configure -without-visit # Configure Trio\_U without the VisIt install*

*# Initialize Trio\_U environment :*

*\$ source bin/Init\_Trio\_U*

*\$ make optim # Build an optimized (-O3 option) version (~15mn)*

*\$ ls \$exec\_opt*

*\$ make debug # Build a debug (-g -O0 option) version (~5mn)*

*\$ ls \$exec\_debug*



# Other possible builds

*# Build a semi optimized binary (option -O3 with asserts)*

\$ make

\$ ls \$exec

*# Build an optimized binary for profiling (option -pg -O3)*

\$ make prof

\$ ls \$exec\_pg

*# Build an optimized binary for test coverage (option -gcov -O3)*

\$ make gcov

\$ ls \$exec\_gcov

# Other basic commands

*# Clean the install*

```
$ cd $TRIO_U_ROOT
```

```
$ make clean
```

*# To run the non regression tests with a binary (by default : \$exec)*

```
$ triou -check all
```

*# To share your work on the Trio\_U shared Git repository*

```
$ cd $TRIO_U_ROOT
```

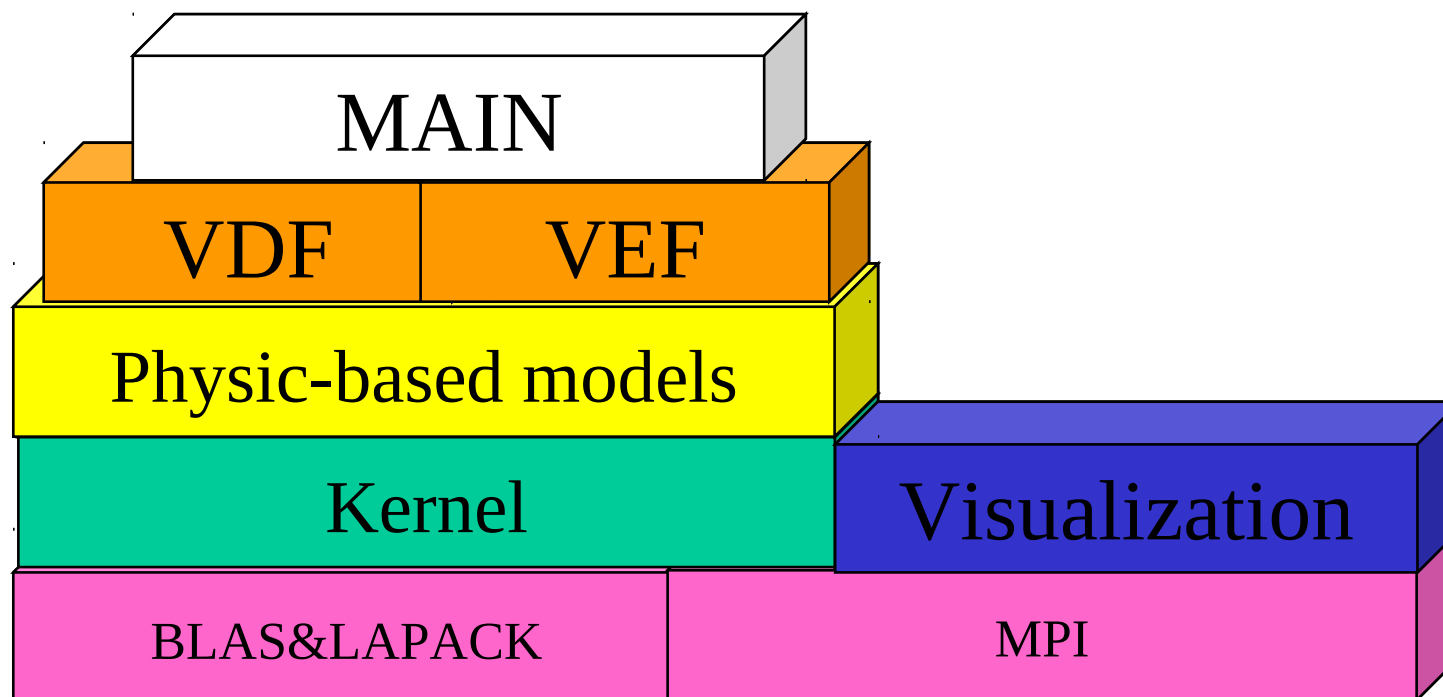
```
$ git commit --all -m"My wonderful work »
```

```
$ git remote add shared git://is223196/shared/trio_u
```

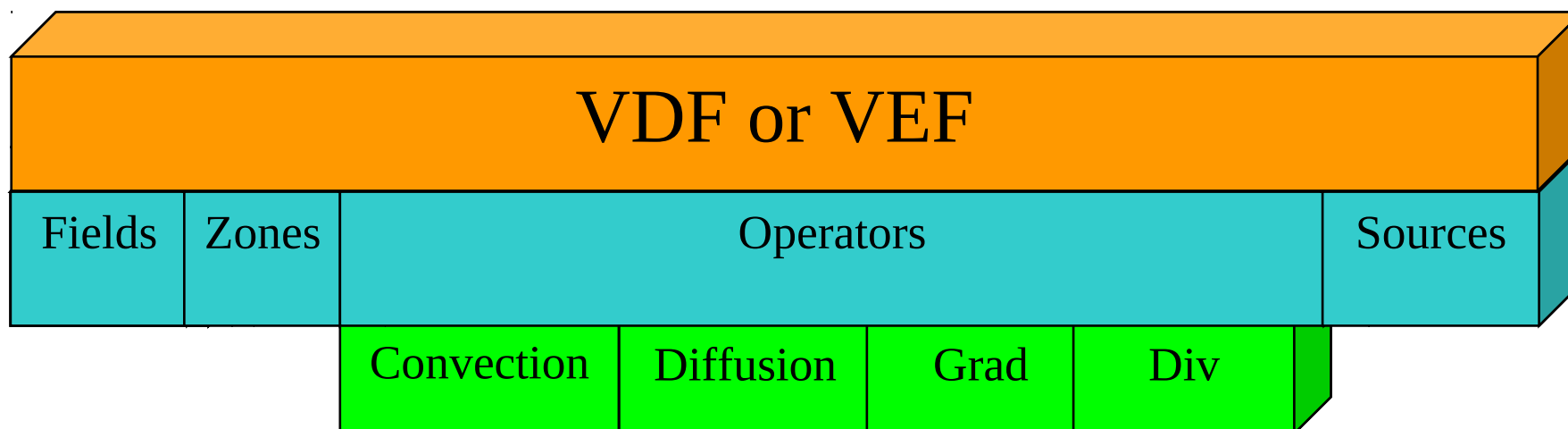
```
$ git push shared
```

# Trio\_U modules

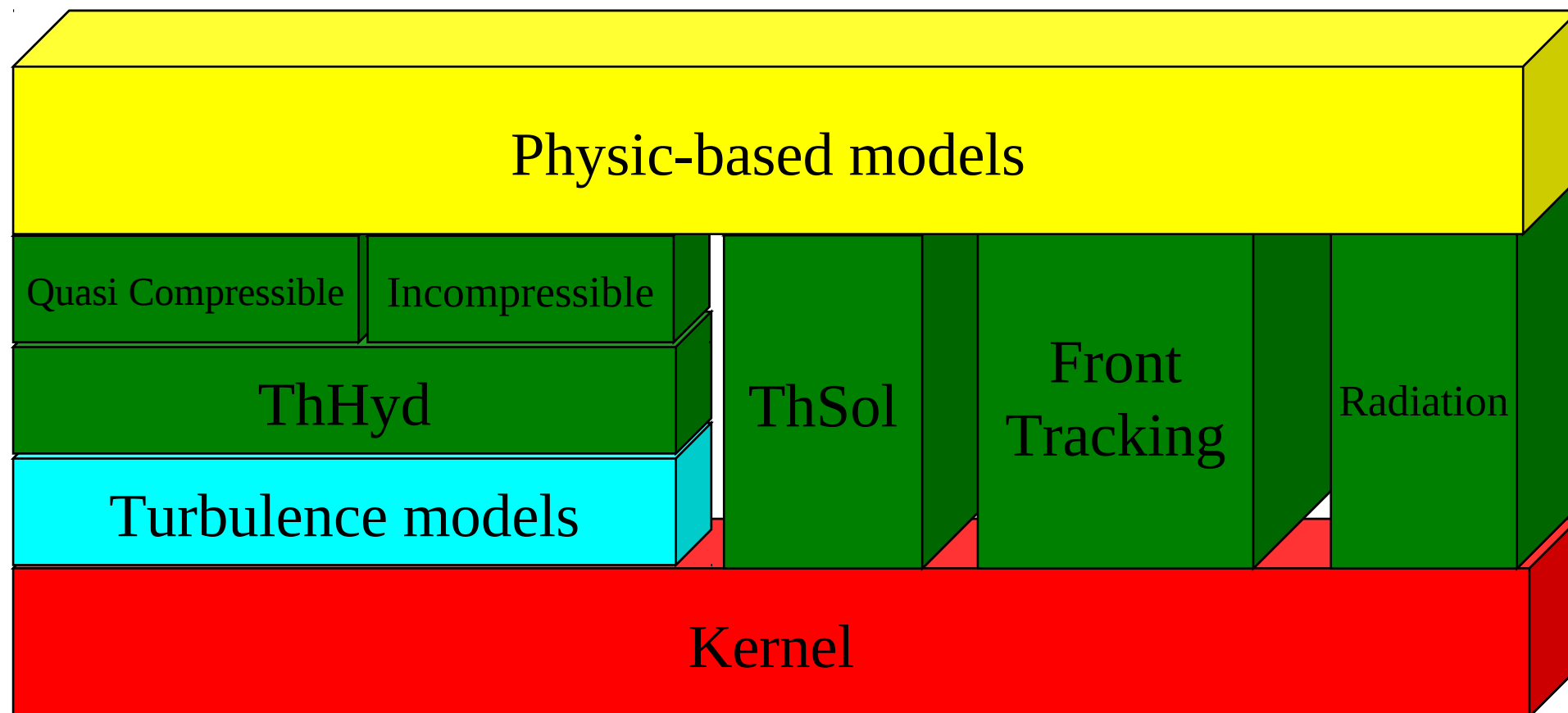
# Trio\_U modules



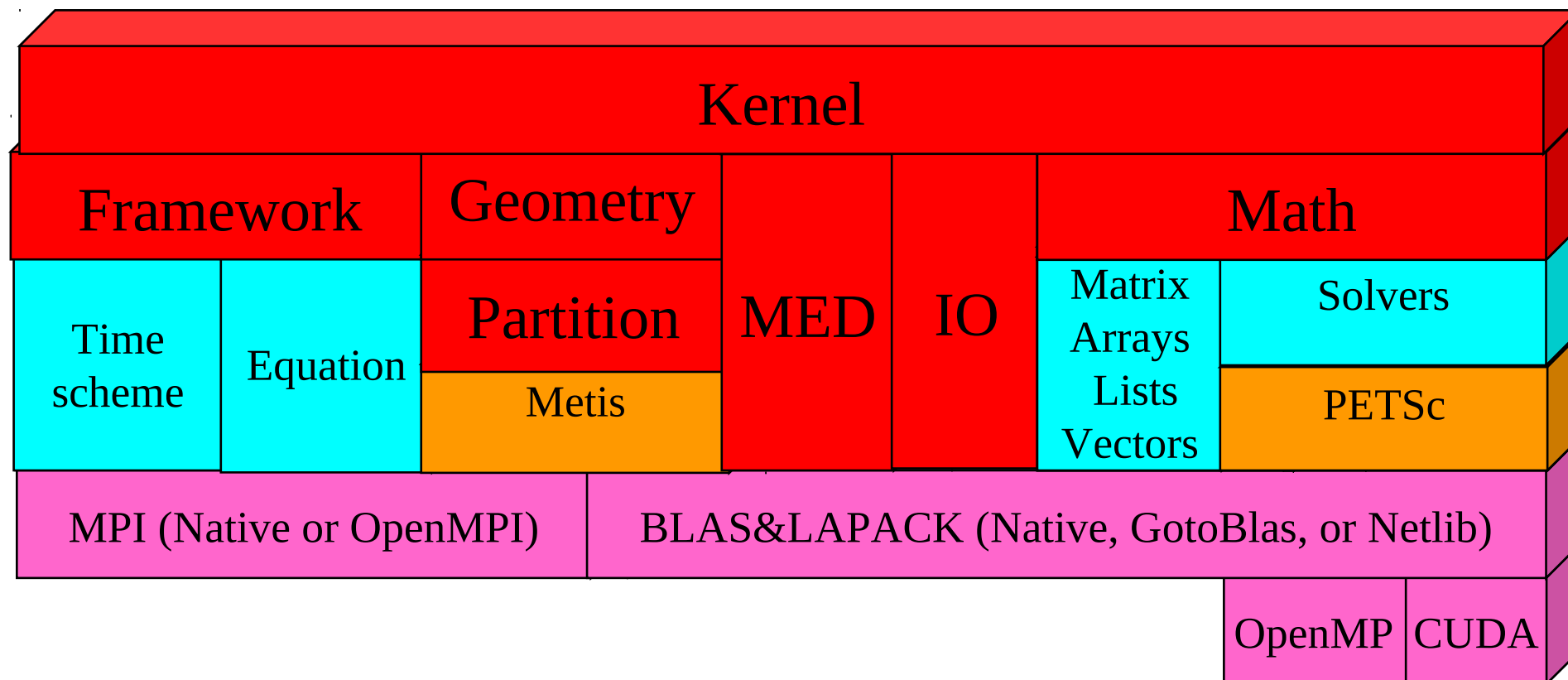
# Discretization modules



# Physics modules



# Kernel module



# Trio\_U/Kernel

- Trio\_U code is made of :
  - 1500 classes
  - Declared in 1500 include files (.h)
  - Implemented in 1500 sources files (.cpp)
  - Within 84 directories
  
- Kernel constitutes 40% of the Trio\_U/Kernel code.
  
- There is a HTML documentation to browse and see the class hierarchy under :  
[\\$TRIO\\_U\\_ROOT/doc/Trio\\_U/html](#)



# Trio\_U/Kernel tests

~40 **Verification forms** to check analytical results under :

`$TRIO_U_ROOT/Validation/Rapports_automatiques/Verification`

~100 **Validation forms** to compare Trio\_U with experimental results or with results from other codes under :

`$TRIO_U_ROOT/Validation/Rapports_automatiques/Validant`

~1900 **Non regression test cases** under :

`$TRIO_U_ROOT/tests`

# Kernel source directories

Under \$TRIO\_U\_ROOT/src/Kernel

./Champs	Generic fields
./Champs_dis	Discretized generic fields
./Cond_Lim	Generic boundary conditions
./Framework	Generic problem, equation, time scheme
./Geometrie	Domain, cell geometry, mesh utilities
./Geometrie/Decoupeur	Partition utilities
./ICoCo	IcoCo coupling interface
./MEDimpl	MED utilities
./Math	Math utilities (arrays...)
./Math/Matrices	Matrix
./Math/SolvSys	Linear system solvers
./Operateurs	Generic operators (gradient,...)
./Schemas_Temps	Time schemes
./Solveurs	Solvers
./Statistiques_temps	Statistical utilities
./Utilitaires	IO, memory, MPI
./VF/Champs	Finite volume fields
./VF/Zones	Finite volume geometry description

# Trio\_U source directories

Under \$TRIO\_U\_ROOT/src

./ALE	ALE method
./Front_tracking_discontin	Discontinuous front tracking method
./Front_tracking_discontin/VDF	VDF implementation
./Front_tracking_discontin/VEF	VEF implementation
./P1NCP1B	VEF discretization
./P1NCP1B/Champs	VEF fields
./P1NCP1B/Cond_Lim	VEF boundary conditions
./P1NCP1B/Operateurs	VEF operators
./P1NCP1B/Solveurs	VEF solvers
./P1NCP1B/Turbulence	VEF turbulence
./P1NCP1B/Zones	VEF geometry description
./Phase_field	Phase_field method
./Phase_field/VDF	VDF implmentation
./Rayonnement	Transparent media radiation model
./Rayonnement/VDF	VDF implementation
./Rayonnement/VEF	VEF implementation
./Rayonnement_semi_transp	Semi transparent media radiation model
./Rayonnement_semi_transp/VDF	VDF implementation
./Rayonnement_semi_transp/VEF	VEF implementation

# Trio\_U source directories

./ThHyd	Thermalhydraulic models
./ThHyd/Chimie	Chemical species model
./ThHyd/Quasi_Compressible	Quasi compressible model
./ThHyd/Quasi_Compressible/Turbulence	Turbulence models
./ThHyd/Quasi_Compressible/VDF	VDF implementation
./ThHyd/Quasi_Compressible/VEF	VEF implementation
./ThHyd/Schemas_Temps	Time schemes
./ThHyd/Turbulence	Turbulence models
./ThHyd/Turbulence/Spectres	Turbulence utilities
./ThSol	Conduction model
./UtilitairesAssemblages	Assembly utilities
./VDF	VDF discretization
./VDF/Axi/Operateurs	Operators in VDF with axis symmetry
./VDF/Axi/Operateurs/Evaluateurs	Flux evaluators in VDF with axis symmetry
./VDF/Axi/Sources	VDF sources term for axis symmetry
./VDF/Axi/Turbulence	VDF turbulence model for axis symmetry
./VDF/Champs	VDF fields
./VDF/Cond_Lim	VDF boundary conditions
./VDF/Elements	VDF cells description
./VDF/Operateurs	VDF operators
./VDF/Operateurs/Evaluateurs	Flux evaluators in VDF
./VDF/Operateurs/Iterateurs	Flux iterators in VDF
./VDF/Solveurs	VDF solvers
./VDF/Sources	VDF source terms
./VDF/Sources/Evaluateurs	Source term evaluators in VDF
./VDF/Sources/Iterateurs	Source term iterators in VDF
./VDF/Turbulence	Turbulence models VDF implementation
./VDF/Zones	VDF geometry description

./VEF	VEF discretization
./VEF/Champs	VEF fields
./VEF/Cond_Lim	VEF boundary
./VEF/Operateurs	VEF operator
./VEF/Solveurs	VEF source terms
./VEF/Sources	Source evaluators in VEF
./VEF/Sources/Evaluateurs	Source iterators in VEF
./VEF/Sources/Iterateurs	Turbulence model implemented in VEF
./VEF/Turbulence	VEF geometry description
./VEF/Zones	Zoom (multi scale simulation)
./Zoom/Algos	Algorithms
./Zoom/Geometrie	Geometry
./Zoom/Kernel	Kernel
./Zoom/Noyau	Operator
./Zoom/Operateurs	VDF implementation
./Zoom/VDF	VDF boundary condition
./Zoom/VDF/Cond_Lim	VDF turbulence model
./Zoom/VDF/Turbulence	Directory with main.cpp
./MAIN	

Note :

Some directories need to be merged  
(e.g : VEF and P1NCP1B) !

# Basic Oriented Object Conception (OOC) concepts used in Trio\_U

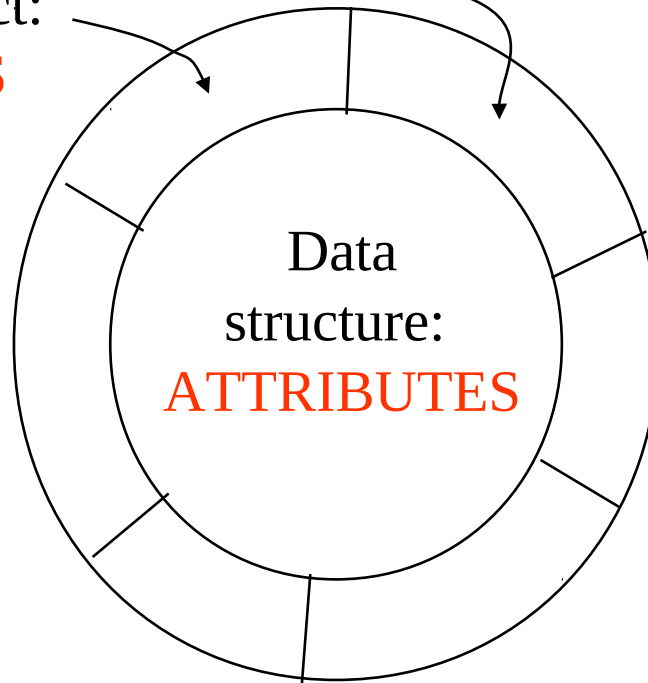
# What are C++ class/object?

- A class is an association of a set of methods and a data structure
- The class defines the plan to create the object
- The object is an instance of the class

Actions which can be done  
by the object:

**METHODS**

Class
Method1() Method2()
Attribute1



Object attributes can only be  
modified by:

- the object itself,
- by other objects using the  
methods of this object.

➡ Data encapsulation

# Data encapsulation

- The aim of data encapsulation is to:
  - hide the **attributes**
  - hide the implementation of the **methods**
- Respecting encapsulation enables a good maintainability. At any time, one can easily :
  - Add/change the implementation of the **methods**
  - Add/change **attributes**with no (or limited) changes to the rest of the code.



# Example of Trio-U objects:

- Problem (Conduction, Hydraulic,...)
- Equation (PDE as  $\partial U / \partial t + \Sigma Op(U) = \Sigma F$ )
- Operator (grad, div, laplacian,...)
- Unknown field (solution of an equation)
- Physical fields ( $\rho, \mu, \lambda, \dots$ )
- Boundary condition (Dirichlet, Neumann, symmetry, ...)
- Time scheme (Euler, Runge Kutta, Implicit, ...)
- Space discretization (VEF, VDF, ...)
- ... and many others at lower level ... Examples:
- Arrays (class DoubleTab for  $A(i,j)$ , class DoubleVect for  $A(i)$ , IntTab, ....)
- String (class Nom)...



# First example: Equation class



See Equation\_base class

## attributes :

- **Nom** nom // A name
- **Ref\_Probleme\_base** mon\_probleme // A reference (link) to a problem
- **Ref\_Schema\_Temps\_base** le\_schema\_en\_temps // A reference to a time scheme
- ...

## methods :

- to access to the attributes:
  - **probleme()** method returns the problem
  - **schema\_temps()** method returns the time scheme
- to evaluate the time derivative of the unknown  $I(x,y,z,t)$  :
  - **derivee\_en\_temps\_inco(DoubleTab& I)** method returns  $\partial I / \partial t = f(I)$
- ...



# Second example: Unknown field class



See **Champ\_Inc\_base** class

## methods :

- **fixer\_nb\_valeurs\_temporelles(int nb)** // To store fields in memory at nb different times
- **valeurs()** // Return the values at the current time  $t(n)$
- **futur(int i=1)** // Return the values at the time  $t(n+i)$
- **passe(int i=1)** // Return the values at the time  $t(n-i)$
- **avancer(int i=1)** // Go to the future (by turning forward the “wheel”)
- **reculer(int i=1)** // Go to the past (by turning backward the “wheel”)
- ...

## attributes :

**Roue\_ptr** les\_valeurs // Pointer to a “wheel” mechanism to manage the different times for the unknown field

# Code example:

```
inconnue.fixer_nb_valeurs_temporelles(2); // 2 memories to store the different times of the unknown inconnue
// present (it is an alias or link) points to U(n) (first memory)
```

```
DoubleTab& present = inconnue.valeurs();
```

```
/* DoubleTab present = inconnue.valeurs(); ← Warning! It is a copy here... */
```

```
DoubleTab& futur = inconnue.futur(); // futur points to the second memory
```

```
// Computation of U(n+1) with an algorithm using U(n) only (one step time scheme)
```

```
// like: futur=present + dt* f(present) <=> U(n+1)=U(n) + dt*f(U(n))
```

```
...
```

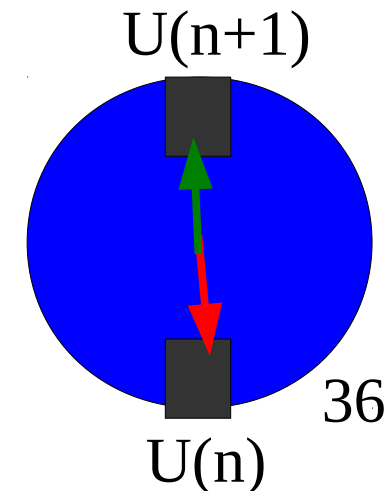
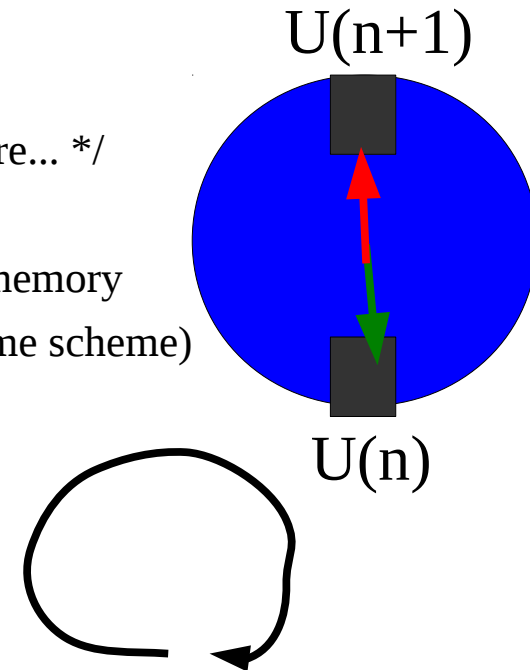
```
// At the end of the time step, we turn the « wheel » with:
```

```
inconnue.avancer();
```

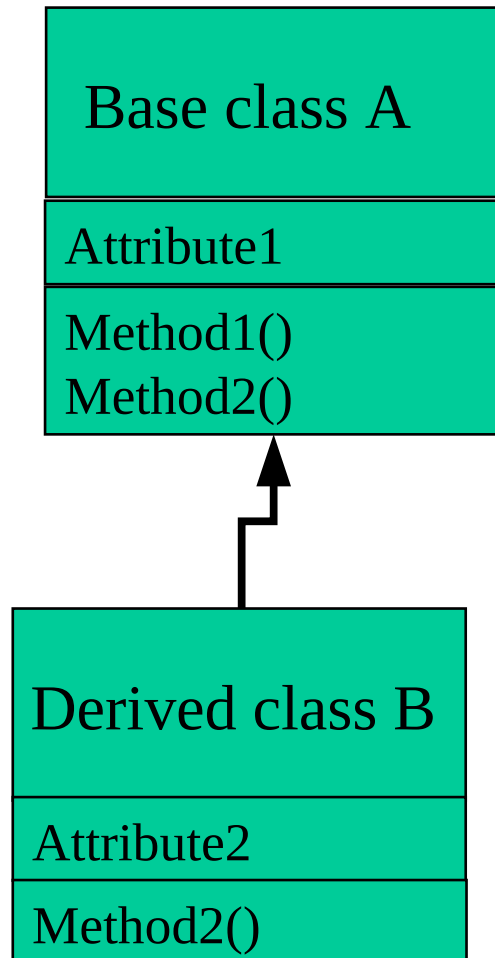
```
// Now valeurs() will return U(n+1) and futur() will return U(n)
```

```
// So during, the next time step, the memory used to store U(n) (now useless)
```

```
// will be overwritten by the storage of U(n+2).
```



# Inheritance



Base class A with 2 methods and 1 attribute.

Derived class B inherits from base class A:

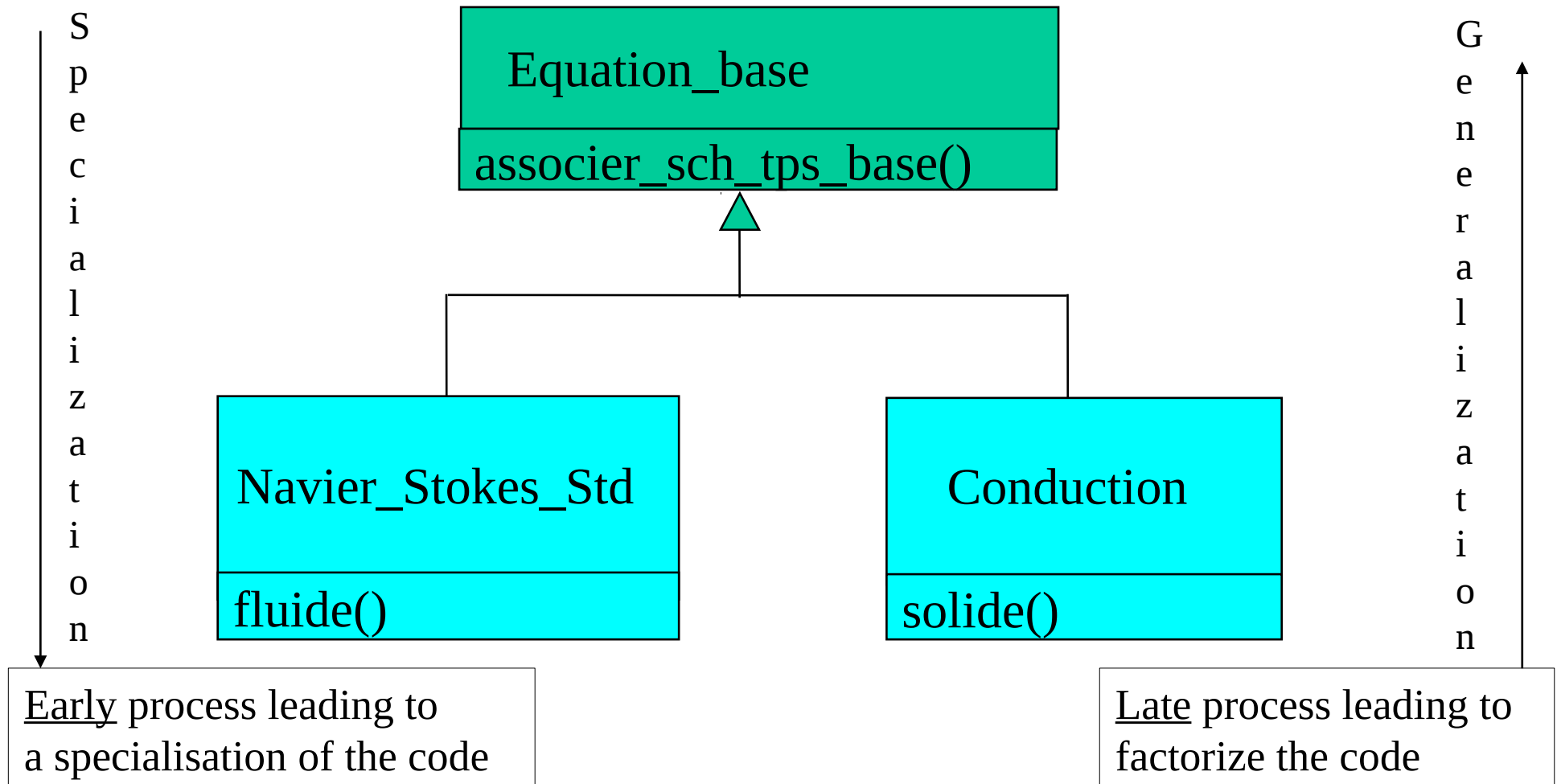
-> Attribute1 and Method1() are **inherited** from the class A

-> B::Method2 method **overloads** A::Method2

# Interest of inheritance

- **Factorization**
  - Identical attributes and methods in different derived classes will be declared and/or implemented once in the base class.
- **Consistency**
  - All the derived classes have, at least, the same interface (methods) than the base class.

# Inheritance example



# Exercise: Use HTML doc

# Browse the Trio\_U ressources index file :

\$ konqueror|firefox [\\$TRIO\\_U\\_ROOT/index.html](#)

# Or :

\$ triou -index

# Select the C++ classes link and look for :

- Inheritance graph of the Navier\_Stokes\_Std **class**
  - Q: How many classes inherits from this class ?
- **Code** file Nom.cpp and the class Nom constructors
  - Q: What is the default value of an object Nom when created ?
- Non const **method** Intab& Zone\_VF::face\_voisins()
  - Q: How many methods in the code use this method ?
- List all the members of the Zone\_VEF **class**
  - Q: In which class is implemented its nb\_elem() method ?



# Polymorphism use in Trio\_U

→ Example of the *derivee\_en\_temps\_inco()* method which implements the calculation of  $F(U)$  in  $\partial U / \partial t = F(U)$ , where  $U$  is the main unknown of the equation

– Static polymorphism (decision is made at the compile time):

```
Navier_Stokes_std eqn;  
eqn.derivee_en_temps_inco();
```

– Dynamic polymorphism (decision is made at the run time):

```
Equation eqn; // Equation is a generic class in Trio_U  
if (...)  
    eqn->typer(Navier_Stokes_std);  
else  
    eqn->typer(Navier_Stokes_Turbulent);  
....  
eqn->derivee_en_temps_inco();
```

# Polymorphism implementation with real and virtual methods

-A real method (default case):

- can be overloaded
- enable only static polymorphism  
→ In the example, A()

-A virtual method:

- can be overloaded
- enable dynamic polymorphism  
→ in the example, B()

-A pure virtual method (abstract method):

- **must** be overloaded (otherwise compilation fails),
- make the class abstract (used for example in base classes),
- enable dynamic polymorphism  
→ In the example, C()

```
class example
{
    public :
    A() ;
    virtual B() ;
    virtual C()=0 ;
};
```



```
class sub_example
{
    public :
    A() ;
    virtual B() ;
    virtual C() ;
};
```

# Virtual method example

```
class Equation_base : public Objet_U
{
public :
    // Evaluate  $\partial U / \partial t$  (returned in F array) for the equation :
    virtual DoubleTab& derivee_en_temps_inco(DoubleTab& F);
    ...
};
```

```
class Navier_Stokes_std : public Equation_base
{
public :
    virtual DoubleTab& derivee_en_temps_inco(DoubleTab& F) ;
};
```

# Navier Stokes equation

Trio\_U equations are basically set under the form :

$$\partial U / \partial t = F(U) = M^{-1} (\sum O p_i(U) + \sum S_i)$$

But for instance, Navier Stokes equations for an incompressible fluid (U velocity, P pressure, M mass, C convection, L diffusion, B divergence,  $B^T$  gradient, S sources):

$$1) BU = 0$$

$$2) M \partial U / \partial t + CU = -B^T P + LU + S$$

Or by inverting 2) by M gives II):

$$II) \partial U / \partial t = -M^{-1} B^T P + M^{-1} (LU - CU + S) \Leftrightarrow \partial U / \partial t = -M^{-1} B^T P + F(U)$$

Then applying  $BU=0$  on II) leads to I):

$$I) BM^{-1} B^T P = BM^{-1} (LU - CU + S)$$

-> One more equation (Poisson) to compute the pressure P and one additional term  $-M^{-1} B^T P$  compared to the equation basic form  $\partial U / \partial t = F(U)$  to compute velocity

# Virtual method example

```
DoubleTab& Equation_base::derivee_en_temps_inco(DoubleTab& F)
{
    //  $\partial U / \partial t = F(U) = M^{-1}(\sum O p_i(U) + \sum S_i)$ 
    F = 0;
    for(int i=0; i<nombre_d_operateurs(); i++)
        operateur(i).ajouter(F);           //  $\sum O p_i(U)$ 
    les_sources.ajouter(F);                 //  $\sum O p_i(U) + \sum S_i$ 
    return solveur_masse.appliquer(F);     //  $M^{-1}(\sum O p_i(U) + \sum S_i)$ 
}
```

**Note: This method is overloaded in the Navier\_Stokes equation class**



# Virtual method example

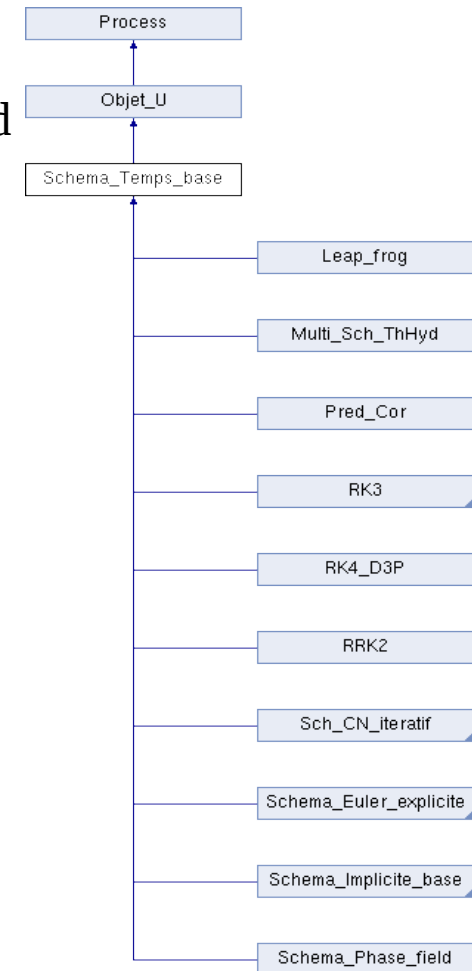
```
DoubleTab& Navier_Stokes_std::derivee_en_temps_inco (DoubleTab& F)
{
    //  $M \partial U / \partial t + \text{grad } P = MF(U) = \sum O p_i(U) + \sum S_j$ 
    //  $\text{div } U = 0 \rightarrow \text{div } M^{-1} \text{grad } P = \text{div } F(U)$ 
    DoubleTab& pression = la_pression.valeurs(); // Storage for P
    DoubleTab& vitesse = la_vitesse.valeurs(); //  $U^n$ 
    DoubleTrav secmem(pression); // Second member
    DoubleTrav gradP(vitesse); // Pressure gradient
    Equation_base::derivee_en_temps_inco(F); //  $F(U)$ 
    divergence.calculer(F, secmem); // secmem =  $\text{div}(F(U))$ 
    solveur_pression.resoudre(secmem, pression); // Solve  $BM^{-1}B^TP = \text{div}(F(U))$ 
    gradient.calculer(pression, gradP); // gradP
    solveur_masse.appliquer(gradP); //  $M^{-1} \text{grad} P$ 
    F -= gradP; //  $F(U) - M^{-1} \text{grad} P$ 
    return F; //  $\partial U / \partial t = F(U) - M^{-1} \text{grad} P$ 
}
```

# Pure virtual method example

*faire\_un\_pas\_de\_temps\_eqn\_base(Equation\_base& equation)* method implements the time scheme to calculate  $U^{n+1}$  for  $\partial U / \partial t = F(U)$  where  $U$  is the main equation unknown

```
class Schema_temps_base : public Objet_U
{
Public :
    virtual int faire_un_pas_de_temps_eqn_base(Equation_base&) =0;
    ...
};
```

```
class Schema_Euler_Explicite : public Schema_temps_base
{
public :
    virtual int faire_un_pas_de_temps_eqn_base(Equation_base &);
};
```



# Pure virtual method example

```
int Schema_Euler_Explicite::faire_un_pas_de_temps_eqn_base(Equation_base& eqn)
{
    //  $\partial U / \partial t = F(U^n) \rightarrow U^{n+1} = U^n + dt * F(U^n)$  for forward Euler scheme
    Champ_Inc& inconnue = eqn.inconnue();           // Equation unknown
    DoubleTab& present = inconnue.valeurs();         // Contains  $U^n$ 
    DoubleTab& futur = inconnue.futur();             // Location to store  $U^{n+1}$ 
    futur = eqn.derivee_en_temps_inco();            //  $F(U^n)$ 
    futur *= dt;                                     //  $dt * F(U^n)$ 
    futur += present;                                //  $dt * F(U^n) + U^n$ 
    return 1;
}
```

Note: These (simplified) example may differ from the current version of the code



# Hello world exercise

- Edit the `$TRIO_U_ROOT/src/MAIN/mon_main.cpp` file and add this lines after “Process::imprimer\_ram\_totale(1);” :

```
std::cout << “Hello World to cout.” << std::endl;  
std::cerr << “Hello World to cerr.” << std::endl;  
Cout << “Hello World to Cout.” << finl;  
Cerr << “Hello World to Cerr.” << finl;  
Process::Journal() << “Hello World to Journal.” << finl;
```

# Hello world exercise

- Rebuild the code:  
`cd $TRIO_U_ROOT`  
`make optim`
- Run the code sequentially  
`cd $TRIO_U_TMP`  
`touch hello.data`  
`trio hello`
- Run the code in parallel and see the differences  
`trio hello 4`

# Hello world exercise

- `Cout <=> std::cout` on the master process only  
Use this output for infos about the physics (convergence, fluxes,...)
- `Cerr <=> std::cerr` on the master process only  
Use this output for warning/errors only
- `finl <=> std::endl + flush()` on the master process
- `Journal()` prints to `datafile_000n.log` files  
Use this output during parallel development to print plumbing infos which would be hidden during later production runs  
During run, this output can be deactivated with:  
`trio hello -journal=0`

# Baltik

## Building Application Linked with Trio\_U Kernel

# Exercise

## Create a Baltik project

*# Initialize Trio\_U/Kernel environment*

```
$ source $TRIO_U_ROOT/bin/Init_Trio_U
```

*# Create your project from a basic project template:*

```
$ mkdir -p ~/test/my_project
```

```
$ cd ~/test/my_project
```

```
$ cp -r $TRIO_U_ROOT/bin/baltik/templates/basic/* .
```

*# Edit your project file project.cfg to specify name, author and executable*

*# Then configure your project:*

```
$ baltik_build_configure
```

```
$ ./configure
```

# Baltik exercise

*# Create a first class and have a look at the 2 files my\_first\_class.h|cpp*

*\$ cd src*

*\$ baltik\_gen\_class my\_first\_class*

*# Build your project:*

*\$ cd ..*

*\$ ./configure # Necessary each time a source file is added to the project*

*\$ make optim*

*# List other options available for the make command:*

*\$ make*

*# Look for more infos here:*

*# \$TRIO\_U\_ROOT/bin/baltik/doc/README.BALTIK*

# The extensive use of macros in Trio\_U

# Trio\_U important points

Trio\_U does not use, for historical reasons:

- Templates
- STL (Standard Template Library)
- Exceptions (until recently)

-Instead of templates, Trio\_U uses macros

-Instead of using STL, Trio\_U defines LIST, VECTORS,...



# Trio\_U important points

Trio\_U hides pointers as much as possible.

You will never see:

```
class A {  
    private:  
    B *b_;  
};
```

But instead:

```
class A {  
    private:  
    REF(B) b_;  
};
```

# Trio\_U important points

Why Trio\_U hides pointers as much as possible?

First case:

```
A::A()
{
    b_ = new B;
    // Initialize b_
    b_ = ...
}
```

```
A::~~A()
    // Delete b_
    delete b_;
}
```

Second case:

```
A::A()
{
    // Just initialize b_
    b_ = ...
}
```

```
A::~~A()
{
    // Nothing to do. b_ is deleted by the
    // destruction of the object REF(B)
}
```

# Trio\_U macros

Macros are widely used to implement plumbing of several features of Trio\_U. For instance:

- To declare and define the class type :
  - base class (**base** macros)
  - instanciated class (**instanciable** macros)
  - generic class (**deriv** macros)
  - associated class (**ref** macros)

# Trio\_U macros

- To define default class constructor/destructors
- To define default class methods like printOn(), readOn() to print/read objects on output/input streams
- To define easily vector (**VECT**) or list (**LIST**) of objects
- For type casting (**sub\_type** & **ref\_cast** macros)

# Four different kind of classes in Trio\_U:

**Base class**

**Instantiate class**

Associated class

Generic class

# Base class

## **Definition:**

A base class is a prototype for other classes.

It is an abstract class, which can't be instantiated.


## **Trio\_U examples:**

Probleme\_**base**      Problem base class


Equation\_**base**      Equation base class

# Base class

## Declaration (.h file)

```
class A_base : public Objet_U
{
     Declare_base (A_base);
    public : ...
    virtual DoubleTab& calculer();
    protected : ...
    private :
        int attribute1;
        B attribute2;
}
```

## Implementation (.cpp file)

```
 Implemente_base(A_base, «A_base», Objet_U);

Entree& A_base::readOn(Entree& is)
{
    is >> attribute1;
    is >> attribute2;
}

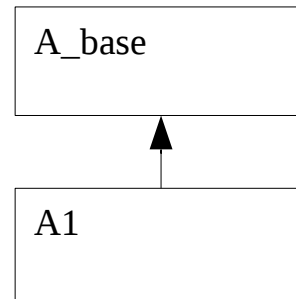
Sortie& A_base::printOn(Sortie& os)
{
    os << attribute1;
    os << attribute2;
}

DoubleTab& A_base::calculer()
{
    ... // que_suis_je() methods returns string « A_base »
}
```

# Instantiate class from a base class

## Declaration (.h file)

```
class A1 : public A_base  
{  
    Declare_instanciable (A1);  
    public : ...  
    protected : ...  
    private : ...  
}
```



## Implementation (.cpp file)

```
Implemente_instanciable(A1, «A1», A_base);
```

```
Entree& A1::readOn(Entree& is)  
{  
    ...  
}
```

```
Sortie& A1::printOn(Sortie& os)  
{  
    ...  
}  
...
```



# readOn - printOn

**printOn** and **readOn** methods are useful to print and read an instantiated object (example, here from A1 class):

```
A1 a;  
EFichier is(« file.txt »); // Trio_U class to read a file  
is >> a ; // Read the 2 attributes from a file  
  
Cerr << a << finl ; // Print the 2 attributes of a  
SFichier os(« newfile.txt ») ;  
os << a ; // Write the 2 attributes of a in a new file
```

# But other macros!

**Declare\_**TYPEOPTION(ClassName);

**Implemente\_**TYPEOPTION(ClassName, »Name »,ParentClassName);

## TYPE:

**base** :For an abstract class

**instanciable** :For an instanciate class

## OPTION:

:Class with a constructor/destructor by default

**\_sans\_constructeur** :Class without a constructor by default (*you* define the constructor)

**\_sans\_destructeur** :Class without a destructor by default (*you* define the destructor)

**\_sans\_constructeur\_ni\_destructeur** :Class without a constructor or a destructor by default (*you* define the constructor/destructor)

# Type casting

## `sub_type` and `ref_cast` macros

`sub_type`(classA,B) : useful to check that a cast is possible  $\Leftrightarrow$  is the class of the object B a derived class of classA ?

`ref_cast`(classA,B) : cast the object B in a classA type object or produces an error if object B is not from a derived class of classA.

# Type casting

## `sub_type` and `ref_cast` macros

### `Solv_Petsc.cpp` example :

```
Int Solv_Petsc::resoudre_systeme(const Matrice_Base& la_matrice, const DoubleVect& secmem, DoubleVect& solution)
```

...

```
if(sub_type(Matrice_Morse_Sym,la_matrice))
```

```
{
```

```
    const Matrice_Morse_Sym& matrice = ref_cast(Matrice_Morse_Sym,la_matrice);
```

```
    assert(matrice.get_est_definie());
```

```
    Matrice_Morse mat;
```

```
    MorseSymHybToMorse(matrice,mat,secmem,solution);
```

```
    Create_objects(mat,secmem);
```

```
}
```

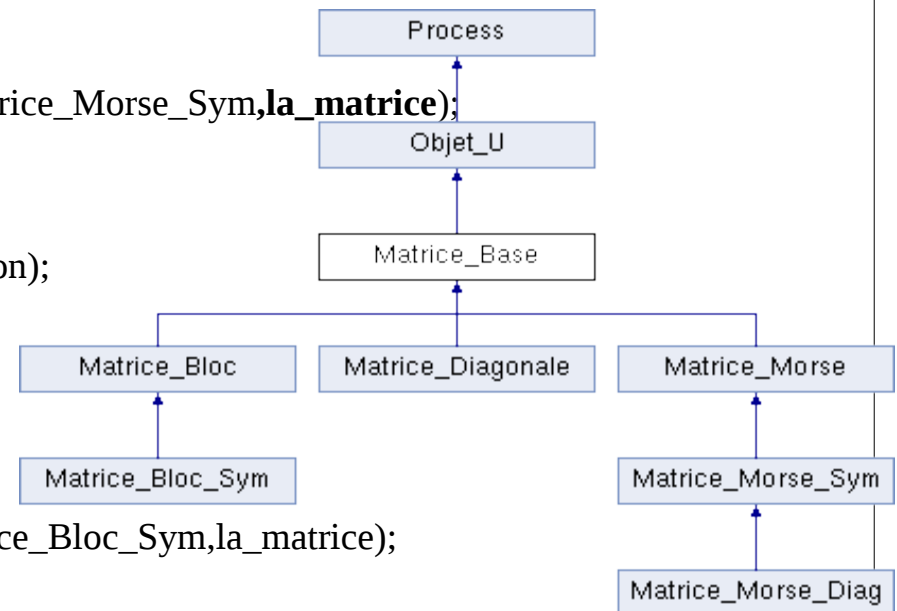
```
else if(sub_type(Matrice_Bloc_Sym,la_matrice))
```

```
{
```

```
    const Matrice_Bloc_Sym& matrice = ref_cast(Matrice_Bloc_Sym,la_matrice);
```

```
    Matrice_Morse_Sym mat_sym;
```

...



Interpretors keywords  
in the data file  
Eg: **Read** keyword  
to read an Object

# Which method is called ?

```
Dimension 3  
Conduction pb  
Domaine dom  
...  
Associate pb dom  
...  
Read pb { ... }
```

-**Read** (as other keywords like Associate) are interpretor keywords. They do several tasks on objects specified by their name (e.g. « pb » name of the problem)

-For each Interpretor, the method of the class **Interpretor** is called when the data file is read :

**Interpretor::interpreter(Entree&) { ... }**

-For example : [Lire.cpp](#)

Link between data  
file and the code  
Eg: **Solve** keyword  
to solve a Problem

# Where is solved a problem ?

Dimension 3

Pb\_hydraulique pb

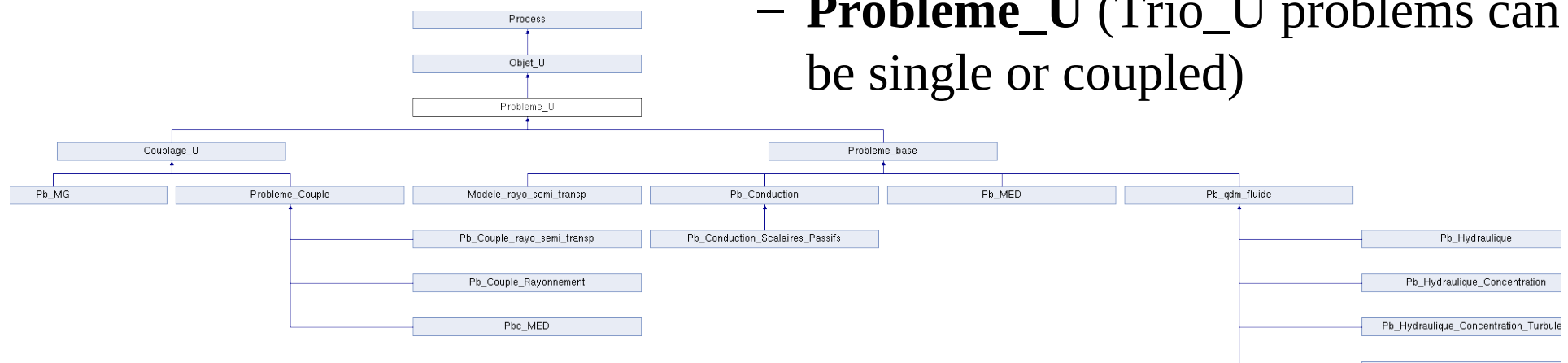
...

Read pb { ... }

**Solve** pb

- The **Solve** interpreter solve the problem
- The object problem is described by a class which inherits from :

- **Probleme\_base** (single base problem)
- **Probleme\_U** (Trio\_U problems can be single or coupled)



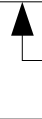


# Resoudre call graph

Dimension 3  
Pb\_hydraulique pb  
...  
Read pb { ... }  
Solve pb

**class Probleme\_U**  
virtual initialize() {}

**class Probleme\_base**  
virtual initialize() { ... }



## Resoudre::Interpreter()

```
{
    Nom problem_name;
    is >> problem_name;
    Probleme_U& pb = ref_cast(Probleme_U,
        objet(problem_name));
    pb.initialize();
    pb.run();
    pb.terminate();
}
```



*Notice how an object is retrieved from its name (objet() method).*

## Probleme\_base::initialize()



- Probleme\_base::preparer\_calcul()
  - milieu().initialiser()
  - Loop on equation(i).preparer\_calcul()
- Schema\_temps\_base::initialize()

*preparer\_calcul() methods make further initializations  
(eg : set time to 0 in fields)*




## Resoudre::Interpreter()

```
pb.initialize()  
pb.run()  
pb.terminate()
```



### Probleme\_U::run()



```
→ computeTimeStep()           // Call to Probleme_base::computeTimeStep()  
    → schema_temps().computeTimeStep() // Calculate first time step dt(0)  
→ Loop on the time steps until stop:  
    → Probleme_base::InitTimeStep()      // Initialize  
        → schema_temps().initTimeStep(); // Set dt=dt(n), initialize flags & residuals  
        → Loop on equation().initTimeStep(); // Set new time on each unknown & BC  
    → Probleme_U::solveTimeStep()        // Solve  
        → Probleme_base::iterateTimeStep() ; // Loop on each problem for this call  
            → schema_temps().iterateTimeStep() ; // Inside, loop on each equation to compute:  
                → faire_un_pas_de_temps_eqn_base(equation(i)) //  $U(n+1) = U(n) + dt * f(U(n))$   
    → Probleme_base::validateTimeStep()    // Update  
        → Schema_Temps_base::validateTimeStep()  
            → Probleme_base::mettre_a_jour()  
                → Loop on equation(i).mettre_a_jour() // Update each unknown & BC  
                → milieu().mettre_a_jour()           // Update the media  
            → Schema_Temps_base::mettre_a_jour() //  $t(n+1) = t(n) + dt(n)$   
    → computeTimeStep()                  // Prepare next  
        → schema_temps().computeTimeStep()           // Compute next time step dt(n+1)  
    → Probleme_base::postraiter()           // Post process the results
```

## Resoudre::Interpreter()

pb.initialize()

pb.run()

pb.terminate()

## Problem\_U::terminate()

→ Probleme\_base::terminate()

→ Probleme\_base::finir()

→ Loop on postraitement(i).finir()

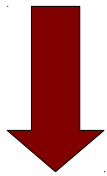
→ Probleme\_base::sauver()

→ Probleme\_base::sauvegarder()

→ Loop on equation(i).sauvegarder() *// Write unknown in backup file*

→ Loop on postraitement(i).sauvegarder()

→ schema\_temps().terminate()



## Know some typical C++ compiler message errors before exercise...

Error : Forward declaration « struct example ...

Error : Invalid use of incomplet type « example ...

-> Missing #include <example.h> where example.h declares the example class.

Error : Cannot declare variable 'a' to be of abstract type 'A' because the following virtual functions are pure within 'A':

-> You need to implement a virtual method declared pure virtual method in the base class

Error : ...

-> ...

# Baltik exercise

# Edit the 2 files:

```
$ cd ~/test/my_project/src
```

```
$ nedit|xemacs|gedit my_first_class.* &
```

# Change the inheritance of the class in order that it inherits not from **Objet\_U** but **Interprete\_geometrique\_base** class instead. It is the base class of all the keywords doing tasks on domains (eg: **Mailler**, **Lire\_fichier**,...).

You will:

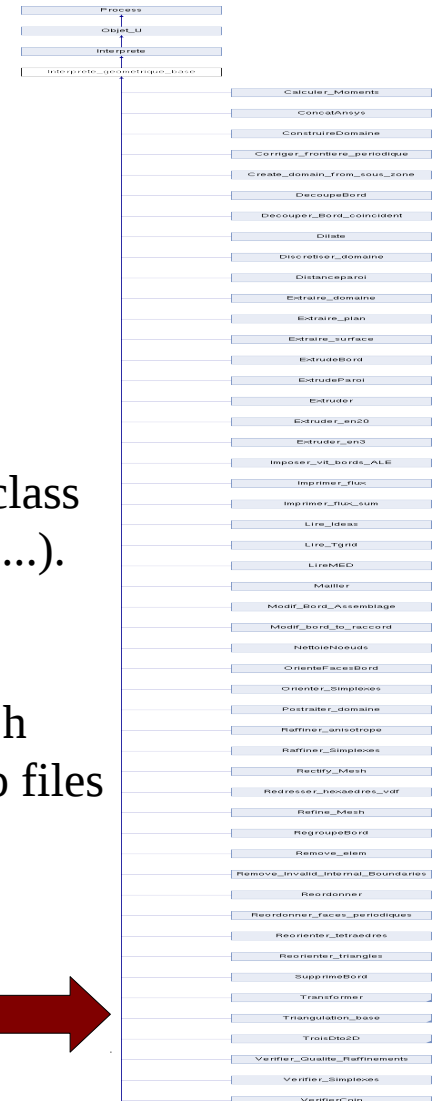
a) add an `#include <Interprete_geometrique_base.h>` in `my_first_class.h`

b) switch **Objet\_U** to **Interprete\_geometrique\_base** in the `.h` and `.cpp` files

c) rebuild your app with:

```
$ cd ~/test/my_project
```

```
$ make debug # An error will occur
```



# Baltik exercise

# You will have an error indicating a pure virtual function (`interpreter_`) should be implemented. Look at the **Interprete\_geometrique\_base** class thanks to HTML documentation from `$TRIO_U_ROOT/index.html` and notice the **interpreter()** method.

This method is called each time a keyword is read in the datafile (eg: **Read\_file** dom dom.geom, **Solve** pb,...)

# Define the public method **interpreter\_(Entree&)** in the include file and implement it (just print a message with Cerr like “My first keyword!”) into the cpp file. **Entree** is a `Trio_U` class to read an input stream (from a file for example):

```
virtual Entree& interpreter_(Entree&);
```

# Rebuild your project and fix your files until the binary of your project is built (named `basic` if you have not changed the name in the `project.cfg` file) :

```
$ cd ~/test/my_project
```

```
$ make debug
```



# Baltik exercise

# Create a test case into the tests/Reference directory of your Baltik project (it should be the directory location of all your test cases for the project):

```
$ mkdir -p ~/test/my_project/tests/Reference/NonRegression
```

```
$ cd ~/test/my_project/tests/Reference/NonRegression
```

```
$ triou -copy Cx
```

```
$ cd Cx
```

```
$ nedit|gedit|xemacs Cx.data
```

# Add into the data file *Cx.data* the keyword **my\_first\_class** just after the line where the problem is discretized, reduce the number of time step to only 1 and run your binary to check that this new keyword is recognized:

```
$ ~/test/my_project/basic Cx
```

# Understand that **Interprete\_geometrique\_base::interpreter()** method is called first, which calls then the **my\_first\_class::interpreter\_()** method.

# Four different kind of classes in Trio\_U:

Base class

Instantiate class

**Associated class**

**Generic class**



# Associations between objects

An object A can have other objects as attributes:

- Either by composition (e.g. of an object from class B) :
  - Object b\_ is created (or destroyed) when an instance from A is created (or destroyed)
- Or by association (e.g. with an object from class C) :
  - Object pointed by c\_ exists independently of any instance of A
  - Implemented by the **REF** macro in Trio\_U:  
**REF**(C) c\_;  $\Leftrightarrow$  C \*c\_;
  - When an instance of A is destroyed, the pointer c\_ is deleted but the pointed object is still in memory:

```
Class A : public Objet_U
{
    public:
        B b_;
        REF(C) c_;
}
```

```
C c;
{
    A a;
    a.c_=&c;
}
```

# Real life example

- Class Car
- Class Tires
- Class Plate\_number

```
Class Car
{
  Tires set_of_tires_;
  Plate_number* number_;
  ...
};
```

In blue, object attributes by composition

In red, object attributes by association

## Equation\_base class example

protected :

```
Nom nom;  
Solveur_Masse solveur_masse;  
Sources les_sources;  
REF(Schema_Temps_base) le_schema_en_temps;  
REF(Zone_dis) la_zone_dis;  
Zone_Cl_dis la_zone_Cl_dis;  
REF(Probleme_base) mon_probleme;  
...
```

In blue, object attributes by composition

In red, object attributes by association

**NOTE** : REF(A) is noted Ref\_A in the HTML documentation

## Associated class (REF)

```
Class A : public Object_U  
{ }
```

```
Class REF(A) : public Ref_  
{ }
```

**Generally declared/implemented in a Ref\_A.h/Ref\_A.cpp files with the 2 macros Declare\_Ref/Implemente\_Ref:**

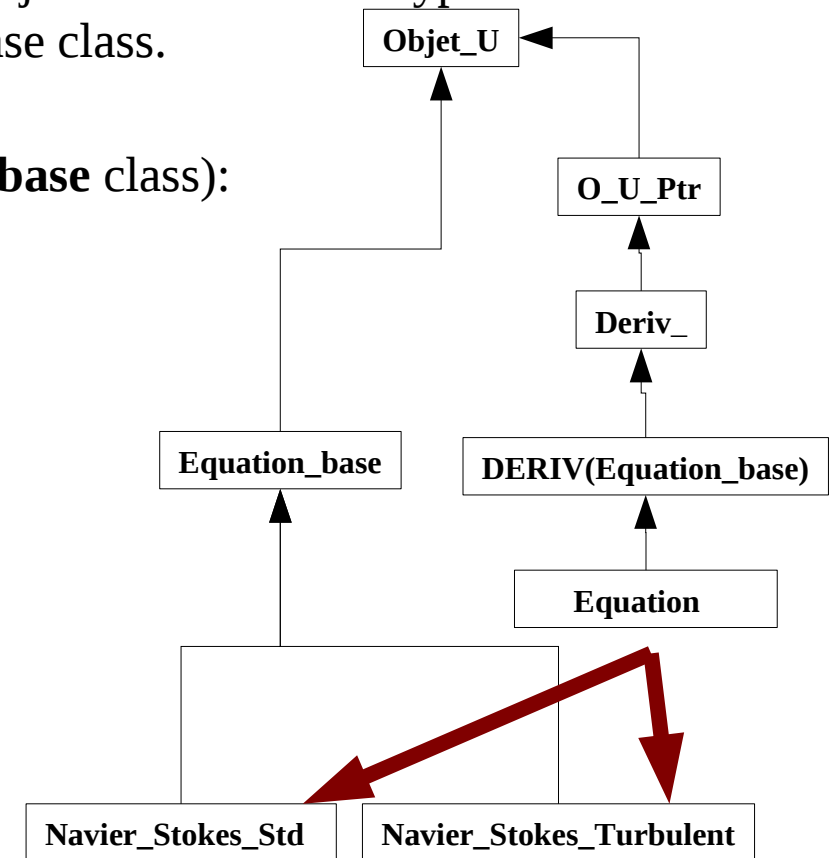
```
#ifndef RefA_inclus  
#define RefA_inclus  
#include <Ref.h>  
class A;  
Declare_ref(A);  
#endif
```

```
#include <Ref_A.h>  
#include <A.h>  
Implemente_ref(A);
```

# Generic class

- Definition: A generic class A is useful to create objects which can be typed at every moment to any object inheriting from A\_base class.
- Example: The **Equation** class (vs the **Equation\_base** class):

```
Equation eqn;
if (...)
    eqn->typer(Navier_Stokes_std);
else
    eqn->typer(Navier_Stokes_Turbulent);
....
eqn->derivee_en_temps_inco();
```



# Generic class (DERIV)

## Declaration (.h file)

```
Declare_deriv(A_base);  
class A : public DERIV(A_base)  
{  
    Declare_instanciable (A);  
    public : ...  
    // Generally inline all the methods  
    DoubleTab& method()  
    protected : ...  
    private : ...  
}  
  
inline DoubleTab& A::method()  
{  
    return valeur().method();  
}
```

## Implementation (.cpp file)

```
Implemente_deriv(A_base);  
Implemente_instanciable(A, « A », DERIV(A_base));  
  
Entree& A::readOn(Entree& is)  
{  
    ...  
}  
  
Sortie& A::printOn(Sortie& os)  
{  
    ...  
}  
...
```

# Generic class

- All generic classes have a **valeur()** method to return the the pointed type of the object, which is different of the object type given by the **que\_suis\_je()** method. Example :

**Conduction** cond; // Instanciati class

Cerr << cond.**que\_suis\_je()** << finl ; // Prints « **Conduction** »

**Equation** eqn; // Generic class

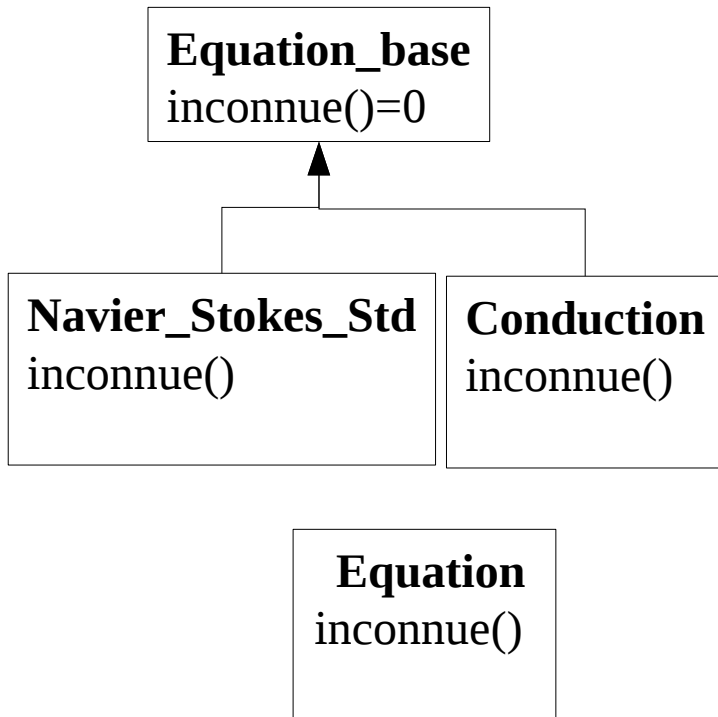
eqn.typer(Conduction) ;

Cerr << eqn.**que\_suis\_je()** << finl ;// Prints « **Equation** »

Cerr << eqn.**valeur()**.que\_suis\_je() << finl; // Prints « **Conduction** »

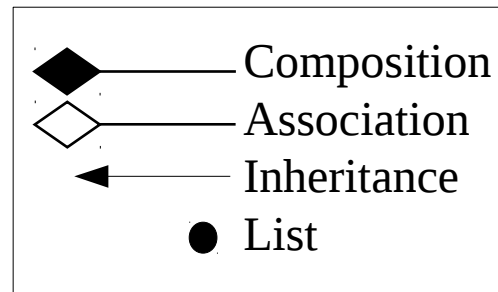
- Often (but not always), hierarchy methods are also coded in generic classes to avoid the use of **.valeur()**. Example :

```
Champ_Inc& Equation::inconnue()
{
    return valeur().inconnue() ;
}
```

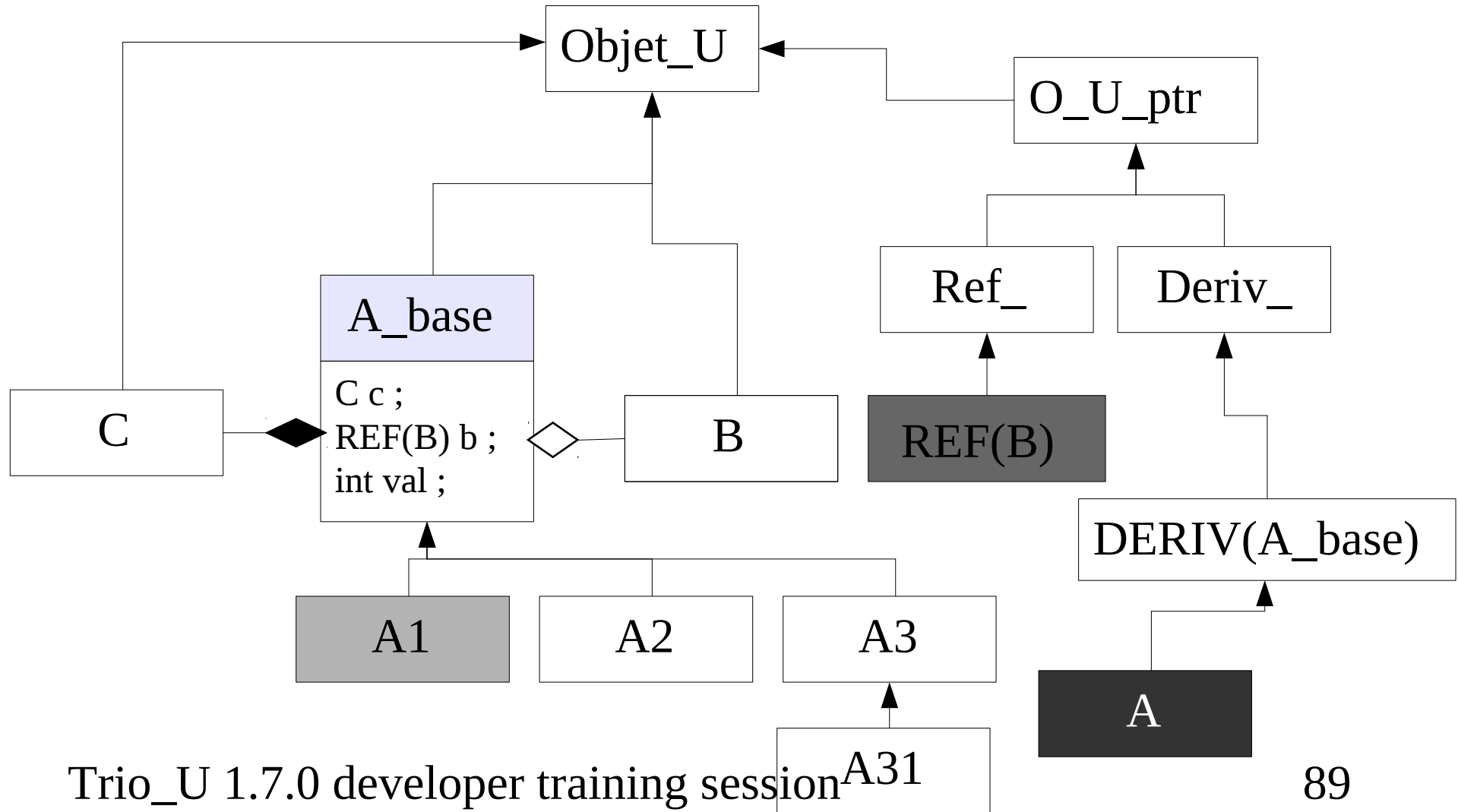
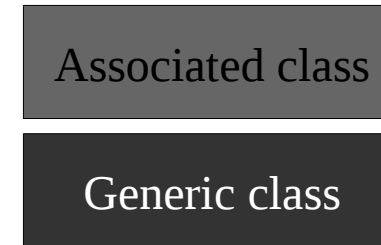
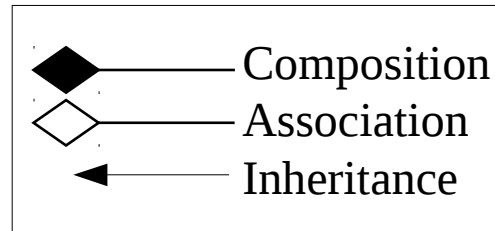
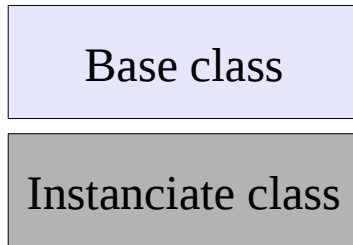


# Hierarchy examples and UML notations

UML (Unified Modeling Language)





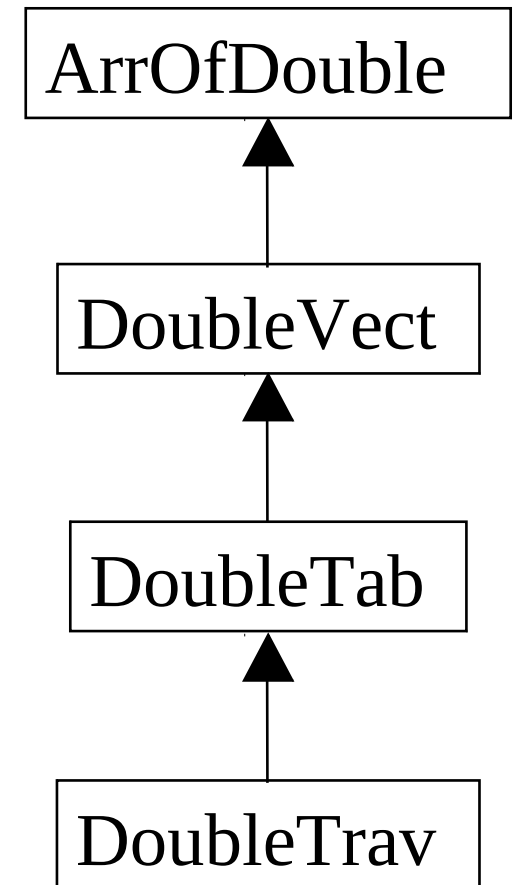


# Exploring the Kernel modules

# Math (Kernel) Part I: Arrays

# Math module

- Array for double :
  - ArrOfDouble A(n)  
→ Basic array, no mechanism to extend data for parallelization
  - DoubleVect A(n)
  - DoubleTab A(n) or A(n,m) or..
  - DoubleTrav A(n)  
→ same than DoubleTab except memory management
- Array for Integer (same but Int instead of Double), example:
  - ArrOfInt, IntVect,...



# Math module

**Difference** between DoubleTab and DoubleTrav

- DoubleTab does a memory allocation/deallocation
- DoubleTrav does a memory allocation but don't deallocate for a future reuse

**Use** Trio\_U arrays cause manage memory for you and detect out of bounds during debug mode runtime.

Example:

```
DoubleTab A(n);  
Cerr << A(n) << finl; // Error detected  
Cerr << A(0,0) << fin; // Error detected
```

# Array examples

// Create and size :

DoubleTab A(n) ;

// Create (A.size\_array()==0) then resize :

DoubleTab A;

if (nb\_comp==1)

    A.resize(n) ;

else

    A.resize(n,2) ;

# Array examples

// Initialize an array :

DoubleTab A(n) ; // A(i)=0.0

DoubleTab A(n,1.0) ; // A(i)=1.0

DoubleTab A(n,1) ; // A(i,0)=0.0

DoubleTab C(n) ;

C=1 ; // C(i)=1.0

DoubleTab B(A) ; // Dimension B and B=A

B+=C ; // B(i)=A(i)+1

# Array examples

```
DoubleTab C ;
```

```
C=B ; // Dimension C according to B and copy values
```

```
C.copy(B, Array_base::COPY_INIT) ; // Same than previous
```

```
DoubleTab C ;
```

```
C.copy(B, Array_base::NOCOPY_NOINIT) ;
```

```
// Dimension C according to B. C(i)=? (uninitialized)
```

```
C.resize_array(n+10, Array_base::COPY_NOINIT) ;
```

```
// C(i<n) is kept. C(n<=i<n+10)= ? (uninitialized)
```



# Array examples

```
DoubleTab A(n,m) ;
```

```
Cerr << A.nb_dim() << finl ;           // 2
```

```
Cerr << A.size() << finl ;             // n*m
```

```
Cerr << A.size_array() << finl ;       // n*m
```

```
Cerr << A.dimension(0) << finl ;       // n
```

```
Cerr << A.dimension(1) << finl ;       // m
```

# How to debug Trio\_U

gdb  
valgrind

# Use gdb tool to debug or understand the code

# To describe all the commands:

**\$ man gdb**

# To debug the Trio\_U binary program compiled with -g:

**\$ gdb \$exec\_debug**

# List of the gdb commands:

run datafile # Run the calculation on the datafile

where or bt # To display the program stack (useful to understand who called what)

up # To move up in the stack

down # To move down in the stack

list # List the source code

cont or c # To continue the calculation after a stop

break class::method # To add a breakpoint on a method of a class

break line # To add a breakpoint on a line of the file once inside a method

break exit # Useful to set a breakpoint just after a Trio\_U error message is printed (before the stack is left)

next or n # Execute next line

step or s # Execute next line and enter in a method/function if any

print var # Print a variable

# Use gdb tool to debug or understand the code

# Specific gdb commands for Trio\_U (macros in a gdb wrapper)

# to dump an array or print array values:

- To dump a DoubleVect : dump array

- To dump a DoubleTab: dumptab array

- To dump a IntVect : dumpint array

- To dump a IntTab: dumpinttab array

- To print tab(i)of a DoubleVect array: print tab.operator()(i) or tab[i]

- To print tab(i,j)of a DoubleTab array : print tab.operator()(i,j) or tab[i,j]

# To debug a parallel calculation with N processes:

mpirun **-gdb** -np N \$exec\_debug datafile N

# Exercise with gdb

Build a debug version of Trio\_U if necessary:

```
cd $TRIO_U_ROOT
```

```
(OPT="";monodir)
```

```
cd tests/Reference/upwind
```

```
gdb $exec_debug
```

```
(gdb) break SSOR::ssor # Stop into the SSOR preconditionner
```

```
(gdb) run
```

```
(gdb) where # Have a look at the stack
```

```
(gdb) n
```

```
(gdb) print tab1 # Or print matrice.tab1_ if "optimized out" message printed
```

```
(gdb) print tab1[10] # Print only a value of an array
```

```
(gdb) dumpint tab1 # Dump the array
```

```
(gdb) print tab1.size_array() # Array size
```



# Exercise with gdb

```
(gdb) up
(gdb) list 100      # Print lines after the 100th line
(gdb) print matrice
(gdb) print matrice.que_suis_je()  # Kind of matrix ?
(gdb) print matrice.que_suis_je().nom_ # Kind of matrix ?
(gdb) up 5 # Move up 5 levels
(gdb) list 900
(gdb) print la_pression.que_suis_je().nom_ # Pressure field
(gdb) print la_pression.valeurs() # Pressure values (DoubleTab)
(gdb) print la_pression.valeurs().nb_dim() # DoubleTab dimension
(gdb) dumptab la_pression.valeurs() # Dump the field values
```

# Use valgrind to find memory bugs

-Valgrind is a memory checker tool: <http://www.valgrind.org>

-You can check a binary with:

\$ **valgrind** \$exec\_debug datafile

Or within the gdb debugger:

\$ gdb -valgrind \$exec\_debug

- It detects uninitialized variables, memory leaks, outbound array values,...

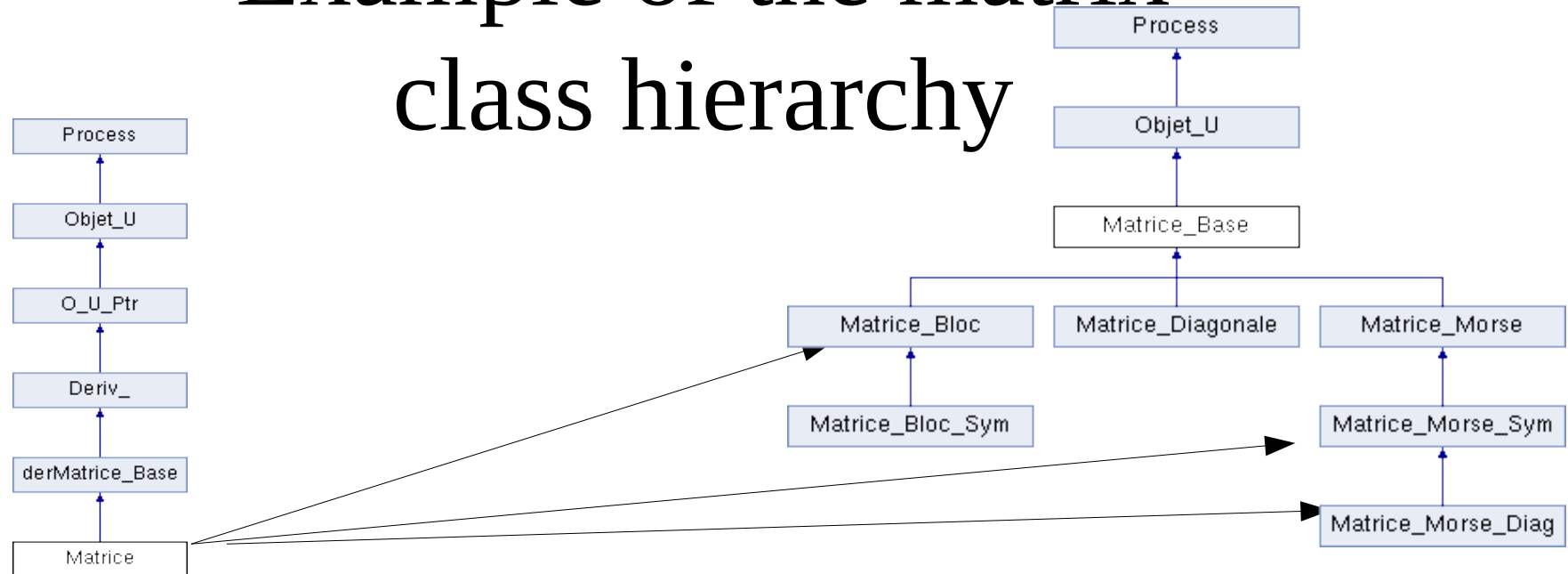
-Trio\_U has 0 errors/warnings/memory leaks according to valgrind on the 2000 non-regression test cases (checked every night). Some errors in third party code (OpenMPI, MUMPS, OpenBlas,...)

# Math (Kernel)

## Part II: Matrix, Vect, List



# Example of the matrix class hierarchy



```

class Matrice_Base : public Objet_U // Base class (and also abstract cause pure virtual method defined)
{ Declare_base(Matrice_Base);
public :
  virtual int ordre() =0 ; ... } ;
class Matrice_Morse : public Matrice_Base // Instanciate class :
{ Declare_instanciable_sans_constructeur(Matrice_Morse); ... } ;
class Matrice : public DERIV(Matrice_Base) // Generic class
{ Declare_instanciable_sans_constructeur(Matrice) ; ... } ;
  
```

## VECT and LIST macros

One can regroup a set of objects of the same kind by using:

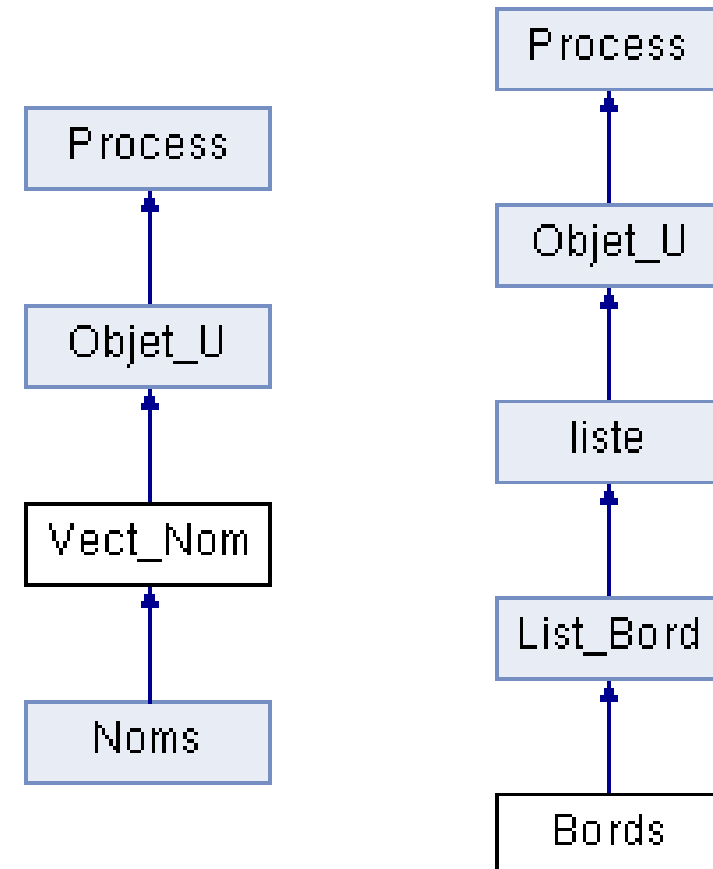
- either VECT, vector of objects
- or LIST, list of objects

Similar interface (search(), add(),...) and performance except for an access to one specific element (LIST slower than VECT in this case)

# Examples of VECT and LIST

Noms  
Bords  
...

VECT(Nom)  
LIST(Bord)



# VECT(class) LIST(class)

## Declaration (.h file)

```
Declare_vect(As);  
class As : public VECT(A)  
{  
    Declare_instanciable (As);  
    public : ...  
    protected : ...  
    private : ...  
}
```

## Implementation (.cpp file)

```
Implemente_vect(As);  
Implemente_instanciable(As, «As»,VECT(A));  
  
Entree& As::readOn(Entree& is)  
{ ... }  
Sortie& As::printOn(Sortie& os)  
{ ... }
```

```
Declare_liste(As);  
class As : public LIST(A)  
{  
    Declare_instanciable (As);  
    public : ...  
    protected : ...  
    private : ...  
}
```

```
Implemente_liste(As);  
Implemente_instanciable(As, «As»,LIST(A));  
  
Entree& As::readOn(Entree& is)  
{ ... }  
Sortie& As::printOn(Sortie& os)  
{ ... }
```

# Practice exercise

Use the HTML documentation to see [MacVect.h](#) and have a look at the VECT methods.

→ Find the method names for ??? in the code :

```
Noms StudentNames ;  
StudentNames.???(3) ;  
StudentNames[0]=... ;StudentNames[1]=... ;StudentNames[2]=... ;  
int number = StudentNames.??? (« Betty » ) ;  
Nom NewStudent (« Bart » ) ;  
StudentNames.??? (NewStudent) ;
```

# Read the data file

The class **Param** use is the recommended choice to read parameters in the data file:

```
#include <Param.h>
Entree& A::readOn(Entree& is)
{
    Nom opt;
    int dim;
    Cerr << « Reading parameters of A from a stream (cin or file) » << finl;
    Param param(que_suis_je());
    // Register parameters to be read:
    param.ajouter("option",&opt);
    param.ajouter("dimension",&dim,Param::REQUIRED); // Mandatory parameter
    // Read now the parameters from the stream is and produces an error if unknown
    // keyword is read or if brackets are not found at the beginning and the end:
    param.lire_avec_accolades_depuis(is);
    ...
    return is;
}
```

A a  
**Read** a { dimension 3 option fast }

# Baltik exercise

# Add into the `interpreter_(Entree&)` method the read of a domain and some parameters into brackets. In the data file, the syntax will be:

```
my_first_class dom { option 0 } # dom is the domain name #
```

Use the following method to read the name of the domain  
*Interprete\_geometrique\_base::associer\_domaine(Entree&)*. Look the HTML documentation. What is the task of this method?

# To help you, have a look at a *Interprete\_geometrique\_base* sub-classe, for instance [Raffiner\\_anisotrope](#) to see how the domain is read. The datafile syntax is:

```
Raffiner_anisotrope DomainName
```



# Baltik exercise

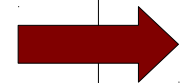
# Then use the **Param** object to read the keyword parameter option in the data file. **Param** use is the recommended choice in this case (even if a lot of current Trio\_U classes still use the old fashion to read parameters), cause it simplifies greatly the coding.

Add `#include <Param.h>` into the cpp file and if help needed, have a look at the `Interprete_geometrique_base` sub-classe **Extruder**. The datafile syntax is :

```
Extruder { domaine DomainName nb_tranches N direction X YZ }
```

Once implementation is finished, add a check at the end of the method `interpreter_(Entree&)` and find how to print the domain name:

```
Cerr << "Option number " << option_number << " has been read on the  
domain named " << ??? << finl;
```





# Baltik exercise

# Build/fix/re-build/run the test case:

```
$ cd ~/test/my_project
```

```
$ make debug
```

```
$ cd tests/Reference/NonRegression/Cx
```

```
$ ~/test/my_project/basic Cx
```

# Terminology/chronology of methods in Trio\_U

## **interpreter()/readOn()**

→ The parameters of the keyword are read

## **associer()**

→ Called by a **Associate** keyword, generally to fill the references (pointer) to other objects (eg : link to an Equation)

## **discretiser()**

→ Called by **Discretize** keyword, complete tasks related to the selected discretization (eg : discretize a field)

## **completer()**

→ All the data file is read, and some initializations are completed now

Loop in the Probleme\_base class on each equation -> [Probleme\\_base.cpp](#)

Loop in Equation\_base class on each operator, discretized boundary condition, sources and time scheme -> [Equation\\_base.cpp](#)

## **preparer\_calcul()**

→ Before the first time step (eg : initialize arrays, set time to 0)

Loop in the Probleme\_base class on each equation -> [Probleme\\_base.cpp](#)

## **calculer()**

→ During the time step, perform the main task of the class

## **mettre\_a\_jour()**

→ At the end of the time step (eg : update time field)

Loop in the Probleme\_base class on each equation -> [Probleme\\_base.cpp](#)

## **postraiter()**

→ At the end of the time step, post process the fields into the result files

*Example :* LES Turbulence model in [Mod\\_turb\\_hyd\\_ss\\_maille.cpp](#)

# Framework (Kernel)

Problem, Domain, Equation, Time steps

# Simple datafile

## Dimension 2

**Domaine** domain **Read\_file** domain file.geom

**Fluide\_Incompressible** media **Read** media { ... }

**Schema\_Euler\_explicite** scheme **Read** scheme { ... }

**VDF** discretization **Read** discretization { ... }

**Pb\_hydraulique** problem

**Associate** problem domain

**Associate** problem media

**Associate** problem scheme

**Discretize** problem discretization

**Read** problem { ... }

**Solve** problem

### 5 objects :

Domain

Media

Scheme

Discretization

Problem

### 5 classes :

Domaine

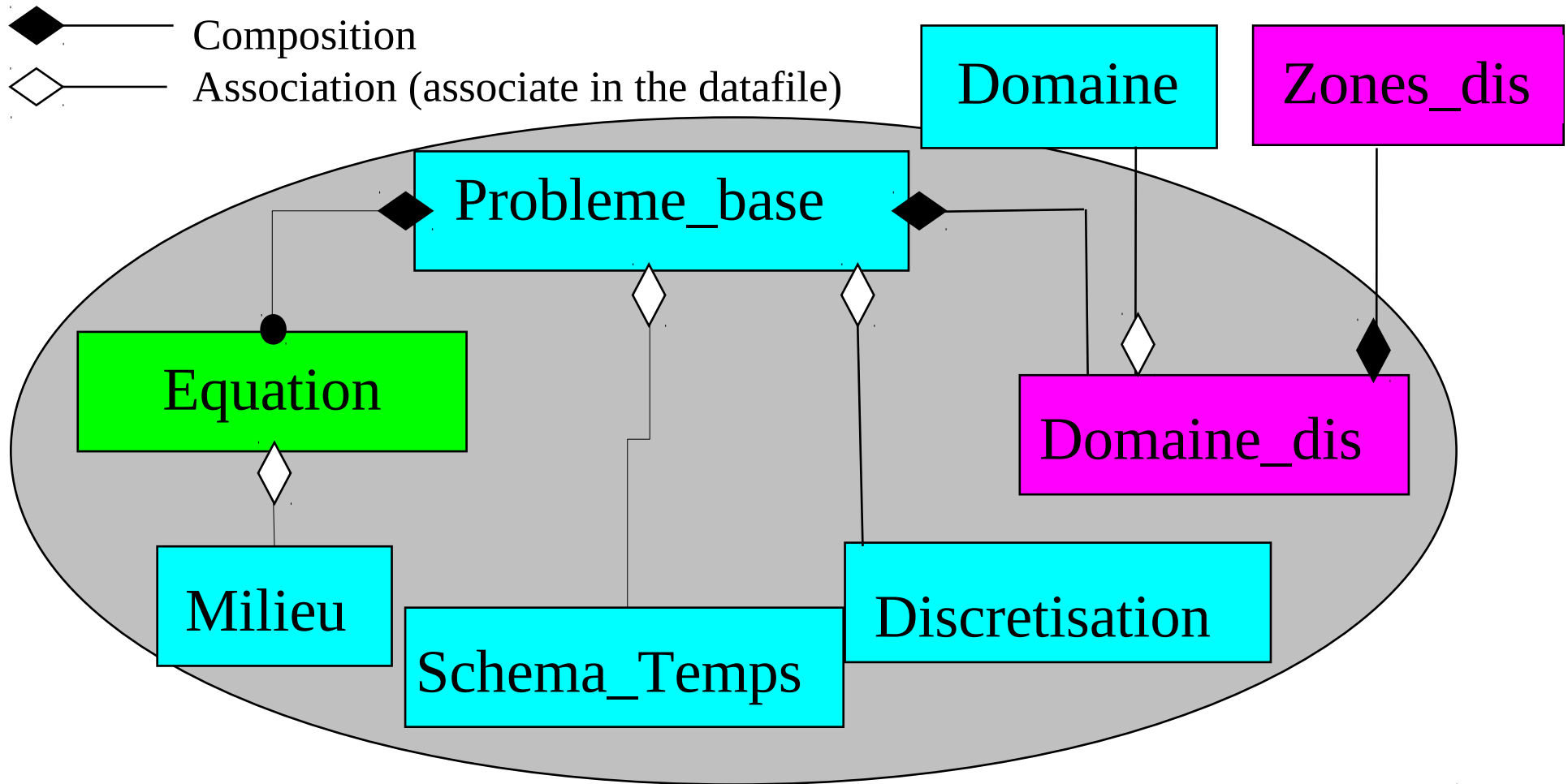
Milieu

Schema\_Temps

Discretisation

Probleme\_base

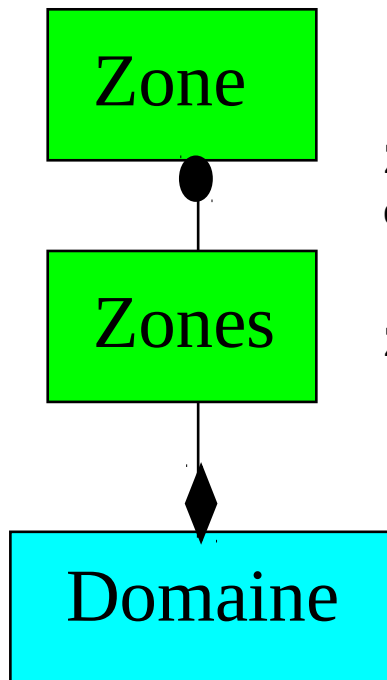
# Problem (Kernel framework)



# Objects creation

- Associated objects should be created before being associated
  - e.g. : **Milieu**, **Schema\_Temps**,...
- Objects by composition are automatically created
  - e.g. : **Equation** and **Domaine\_dis** by the problem
  - What is a **Domaine\_dis** vs a **Domaine**?

# First, Domain and Zone



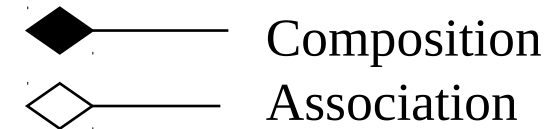
**Domaine** : Spatial domain of resolution of a problem

- Contains the **Zones** and the vertexes (**DoubleTab** sommets) used by the **Zones**

**Zones**: List of meshes to support multi meshes domain (not fully implemented in Trio\_U, so everywhere in the code a **Zones** list has a size of 1).

**Zone** : Is a mesh with cells of same type (eg : tetraedras). It contains :

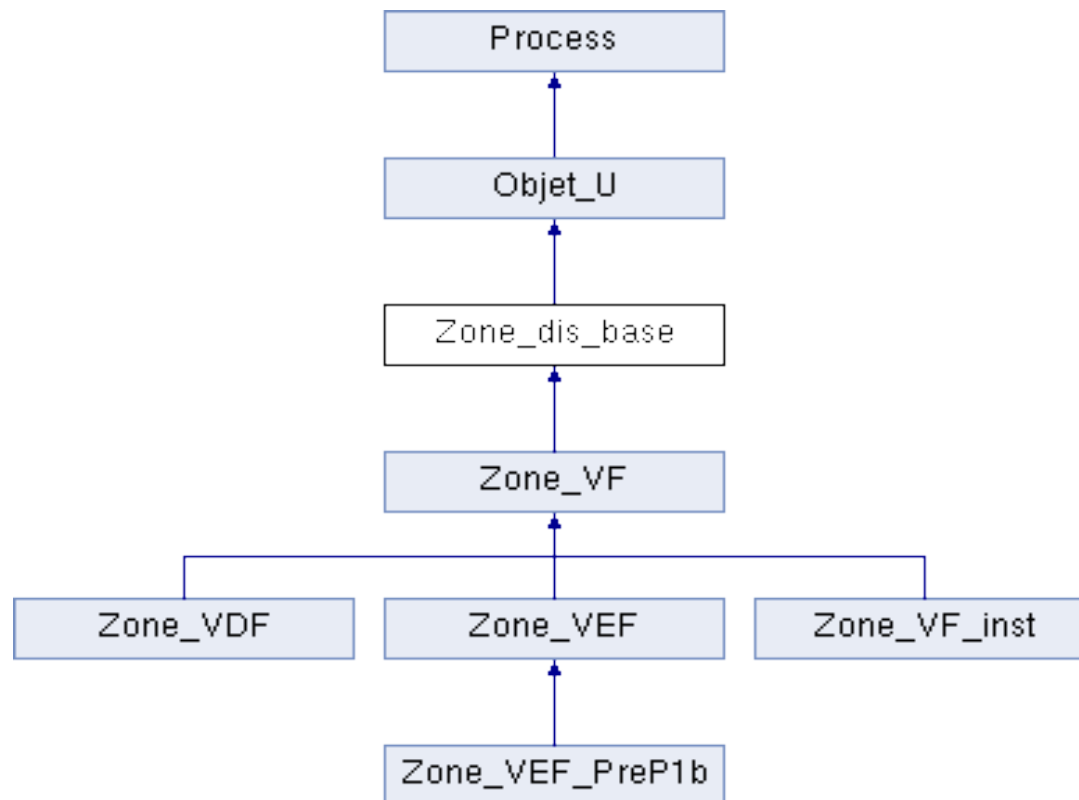
- The cells (**IntTab** mes\_elems)
- The type cell (**elem**)
- The boundaries (« **Bord** » and « **Raccord** ». **Bord** is a boundary, **Raccord** is a boundary where coupling is possible to another domain)
- The boundaries between sub domains for parallelism (« **Joint** »)



**Zone\_dis** is a generic class from Zone\_dis\_base and it depends of the discretization...



# Zone\_dis\_base



**Zone\_VF:** Finite volume description class. Describes control volumes, with xp (center of cells), xv (center of faces)

**Zone\_VDF :** VDF class description with face surfaces, face orientation, ...

**Zone\_VEF :** VEF class description with face normals, face surfaces, ...

**Zone\_VEF\_PreP1B :** Addition to the VEF class (possible edge discretization)

# Baltik exercise

# We are going to try to print informations of the domain boundaries in our current project:

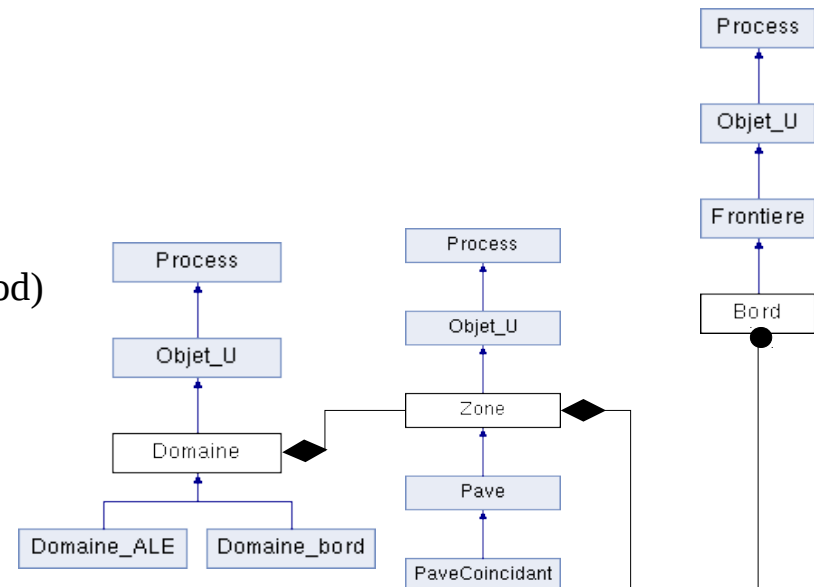
# Edit the *my\_first\_class.cpp* file and add into the *interpreter\_()* method a loop on the boundaries.

# Look for help inside the *Domaine*, *Zone*, *Bord*, *Frontiere* classes into the HTML documentation to access to the:

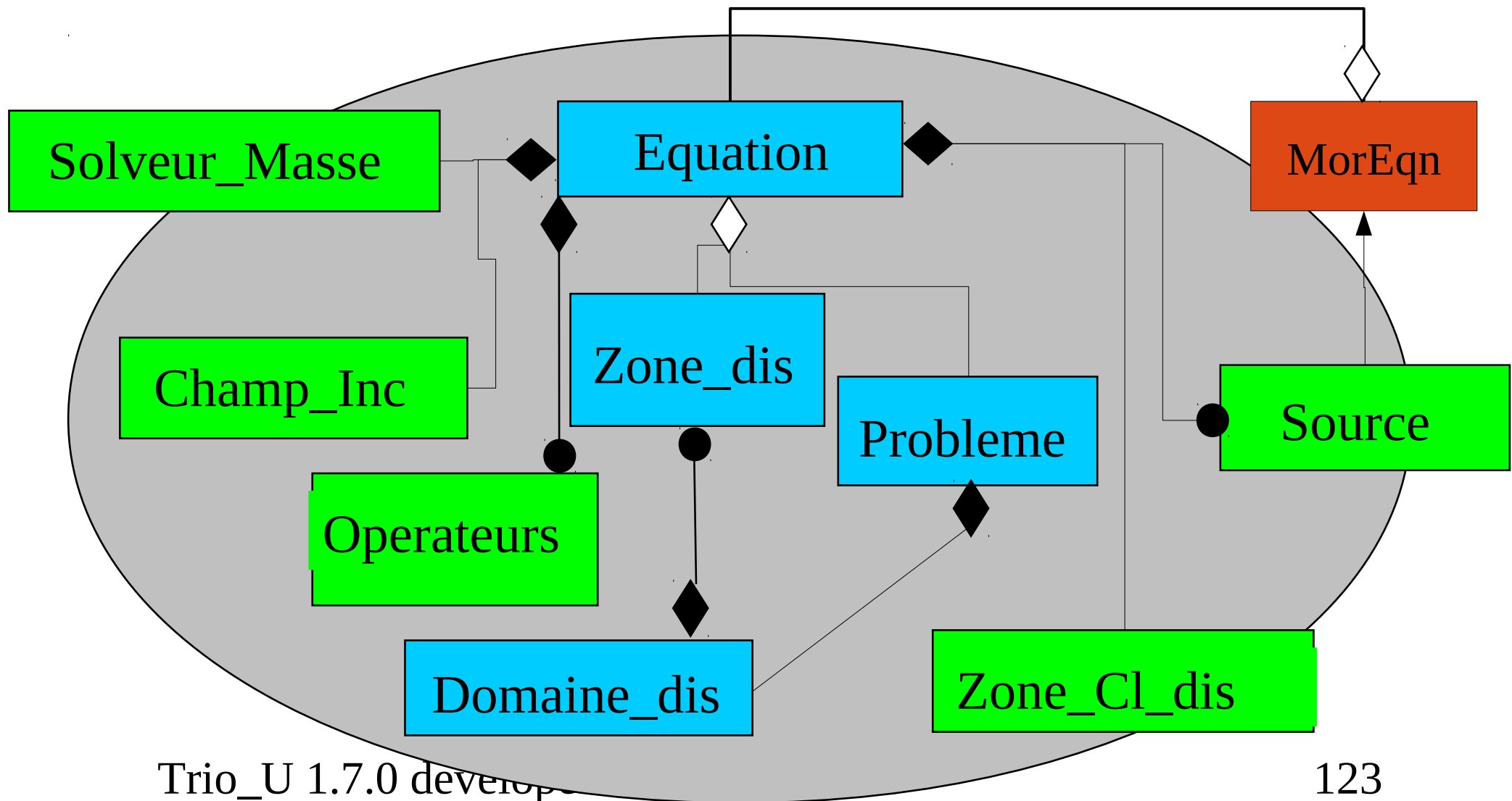
- Number of boundaries (**nb\_bords()** method)
- Boundaries (**bord(int)** method)
- Name of the boundaries (**le\_nom()** method)
- Number of faces of each boundary (**nb\_faces()** method)

# You will print the infos with something like:

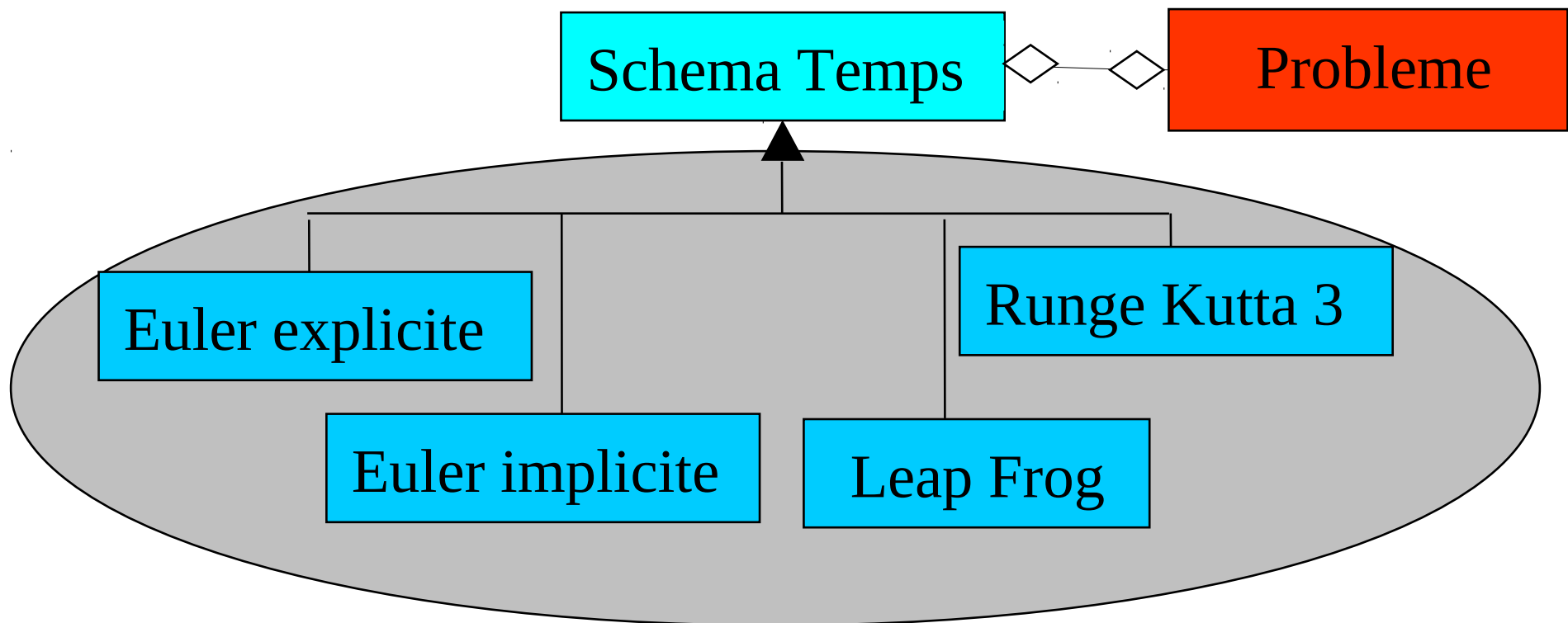
Cerr << "The boundary named " << ??? << " has " << ??? << " faces." << finl;



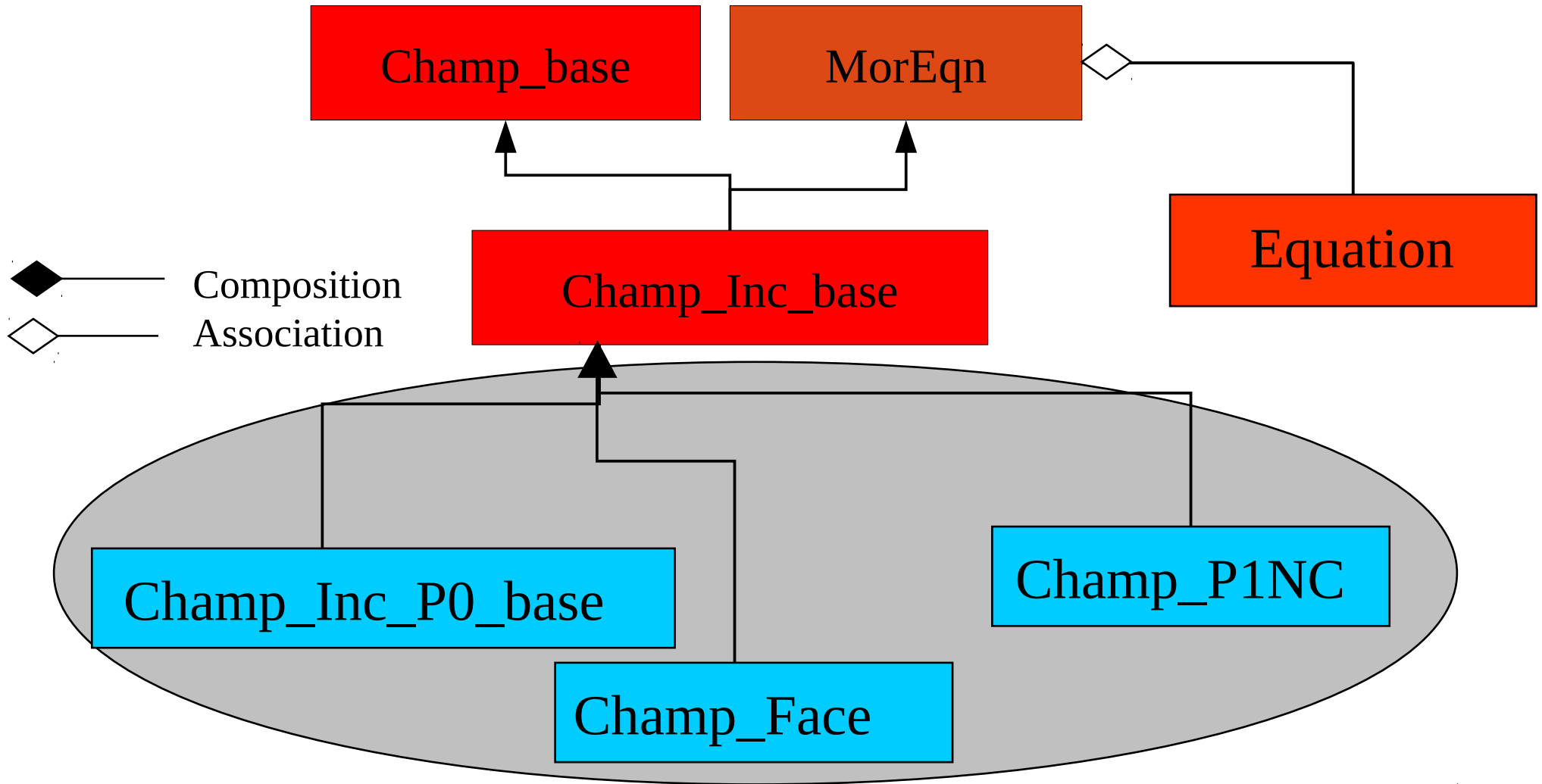
# Equation (Kernel framework)



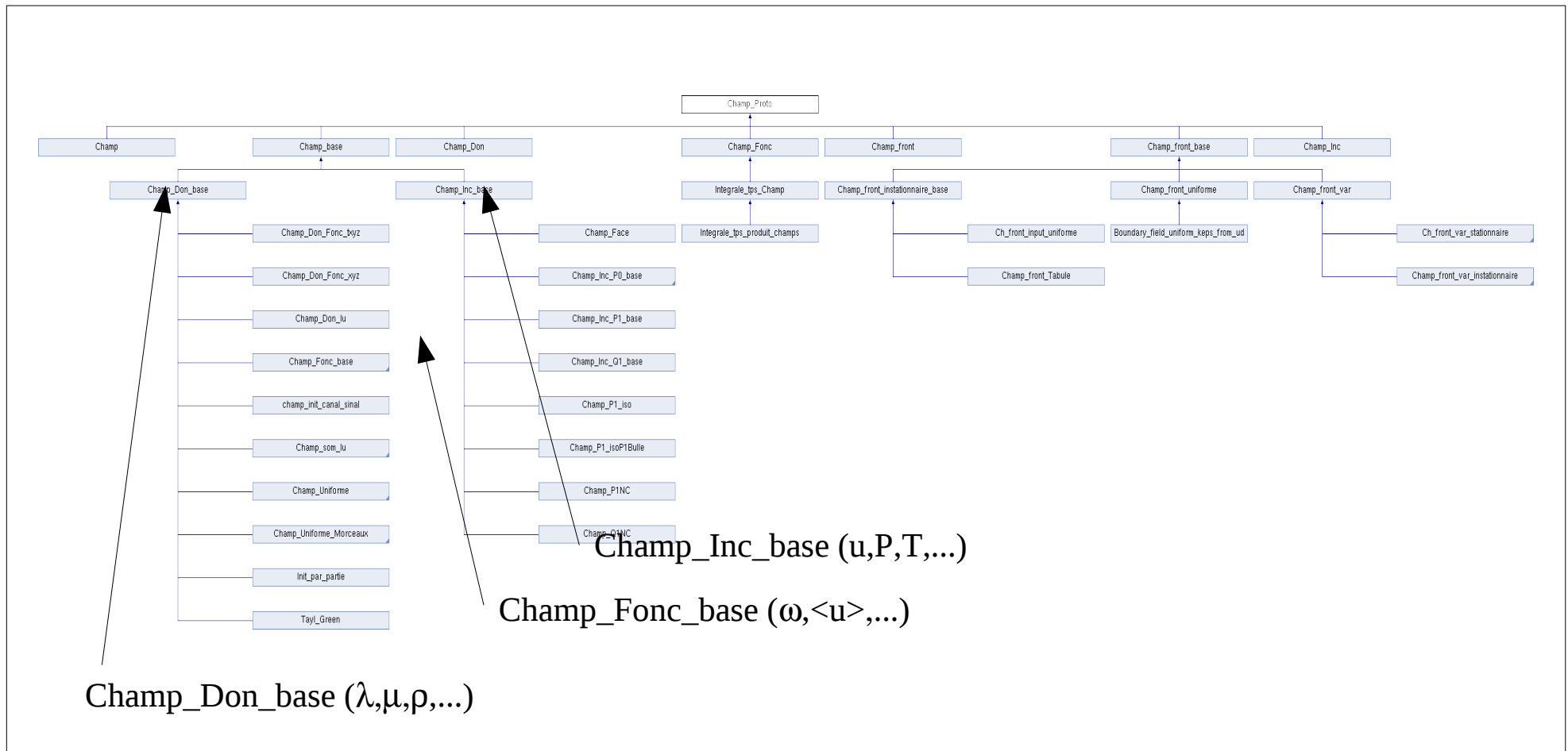
# Time Schemes



# Fields

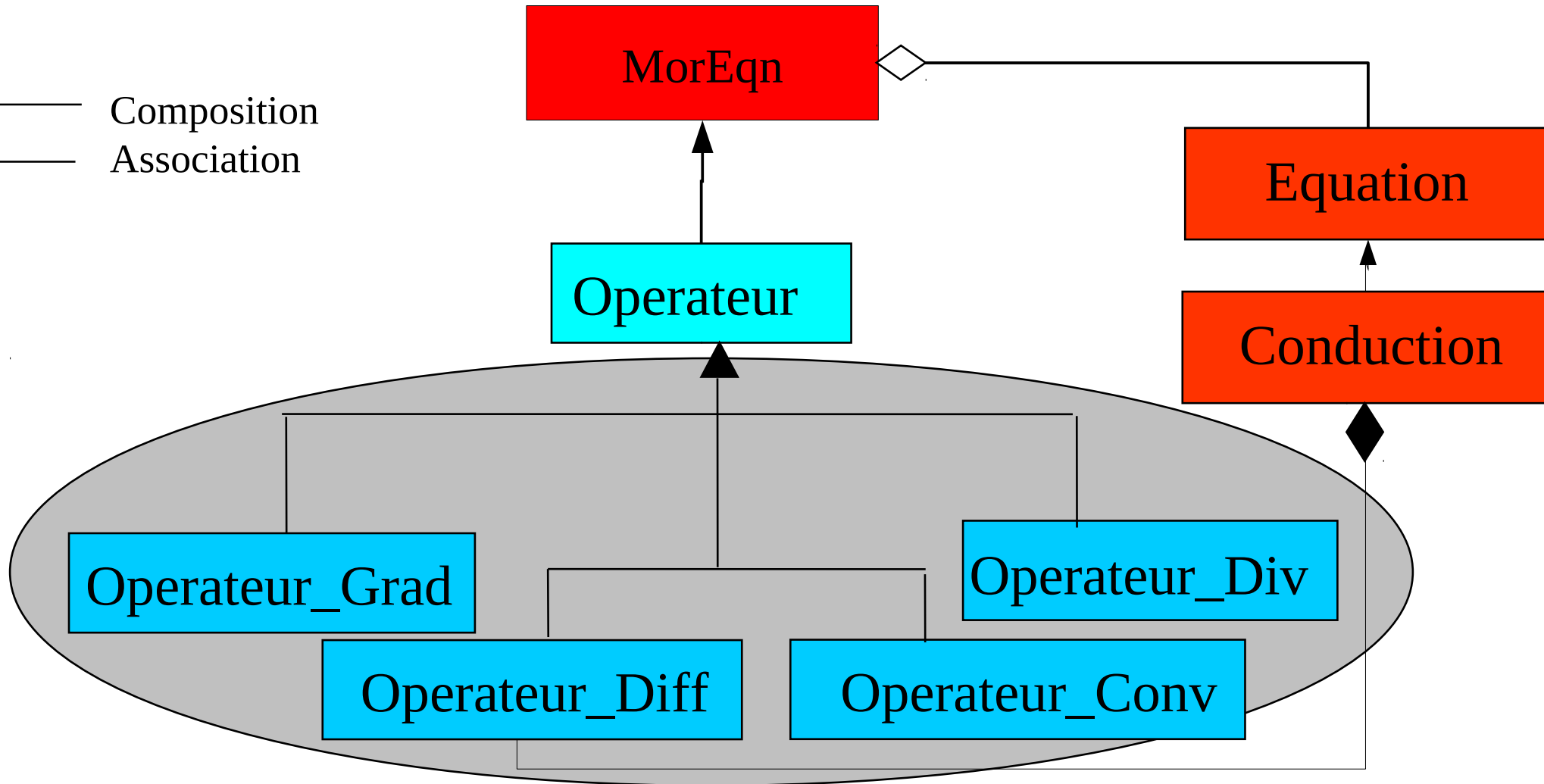


# Field hierarchy



# Operators

◆ Composition  
◇ Association

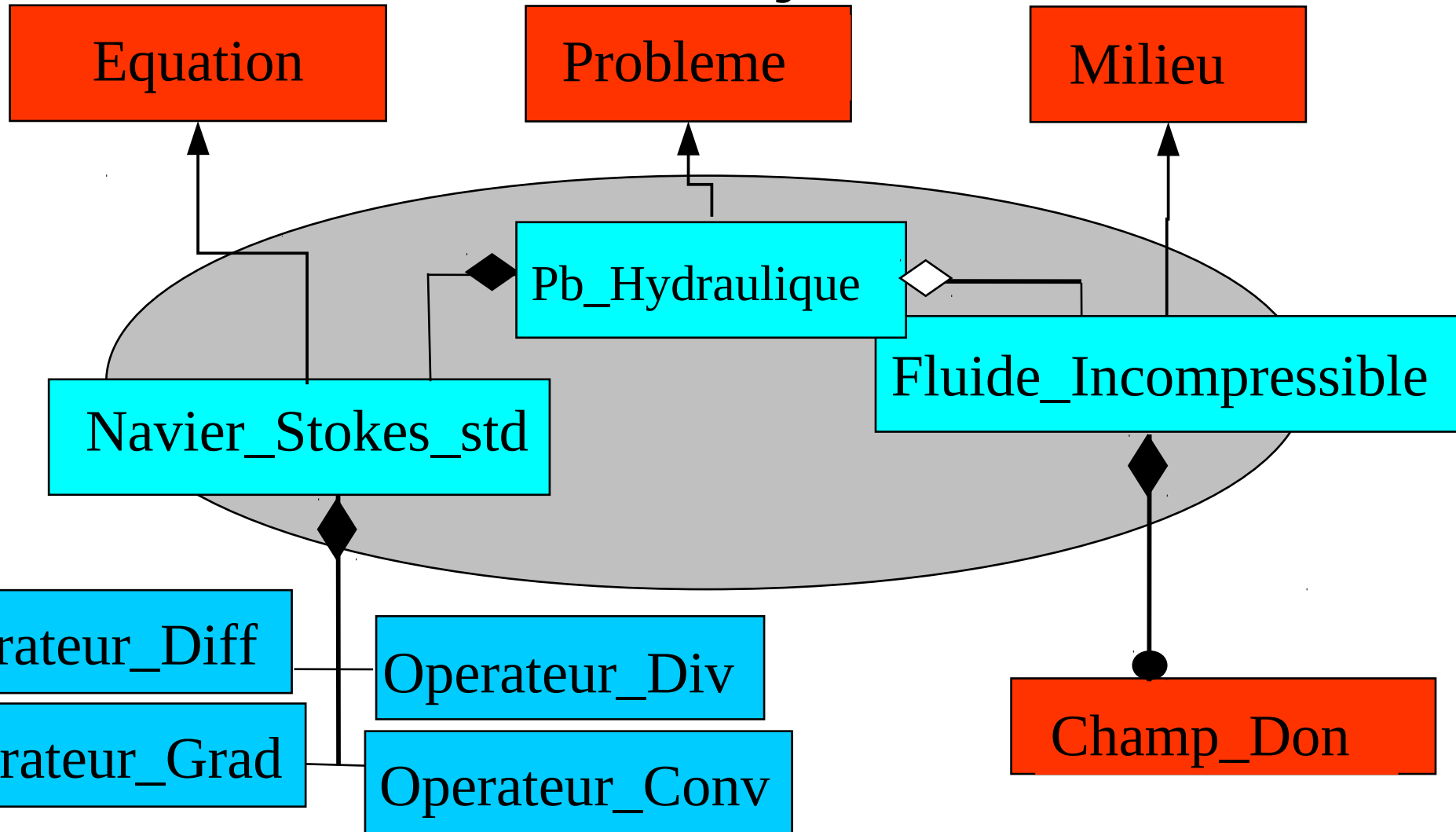


# Trio\_U ThHyd module

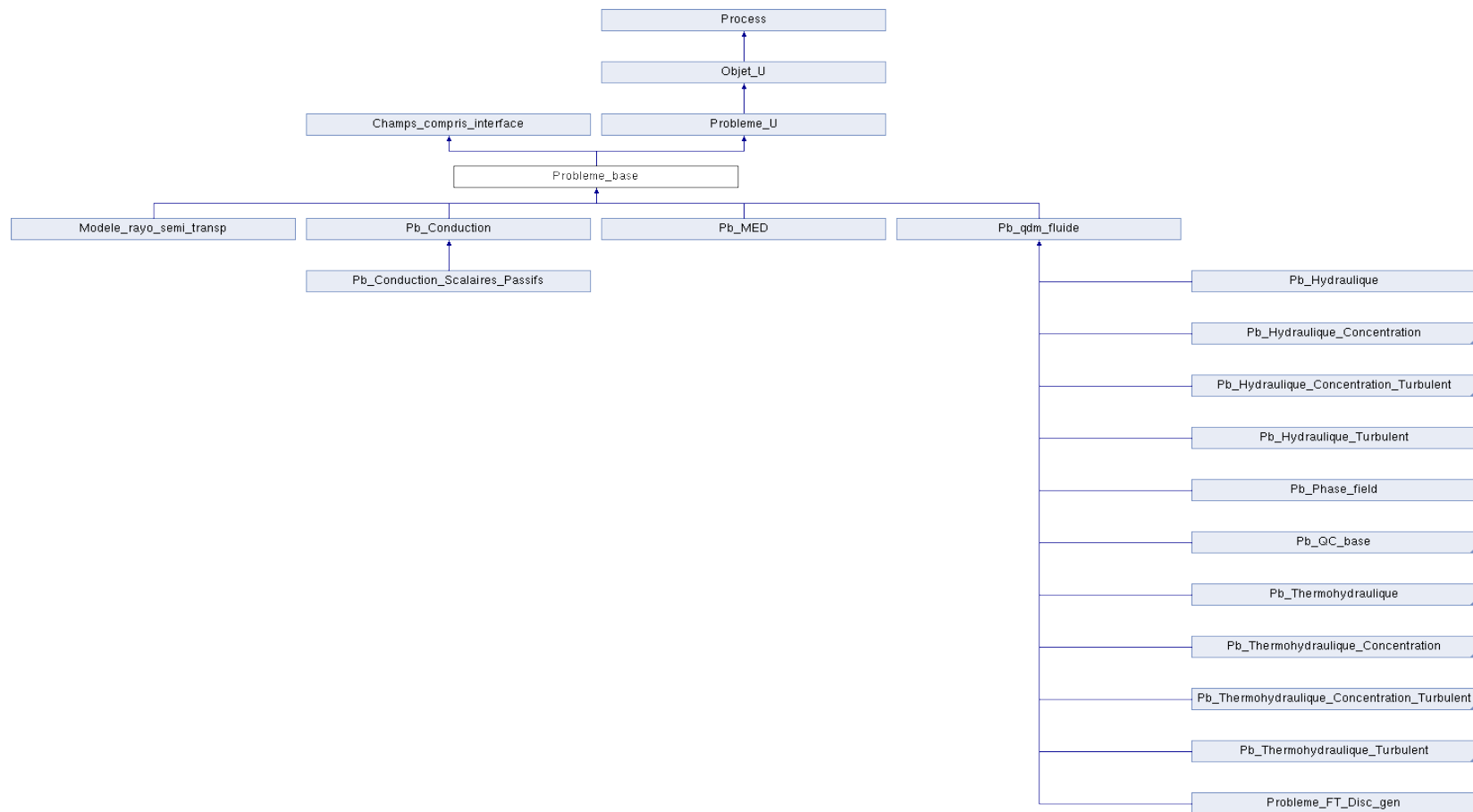
Incompressible Thermalhydraulic



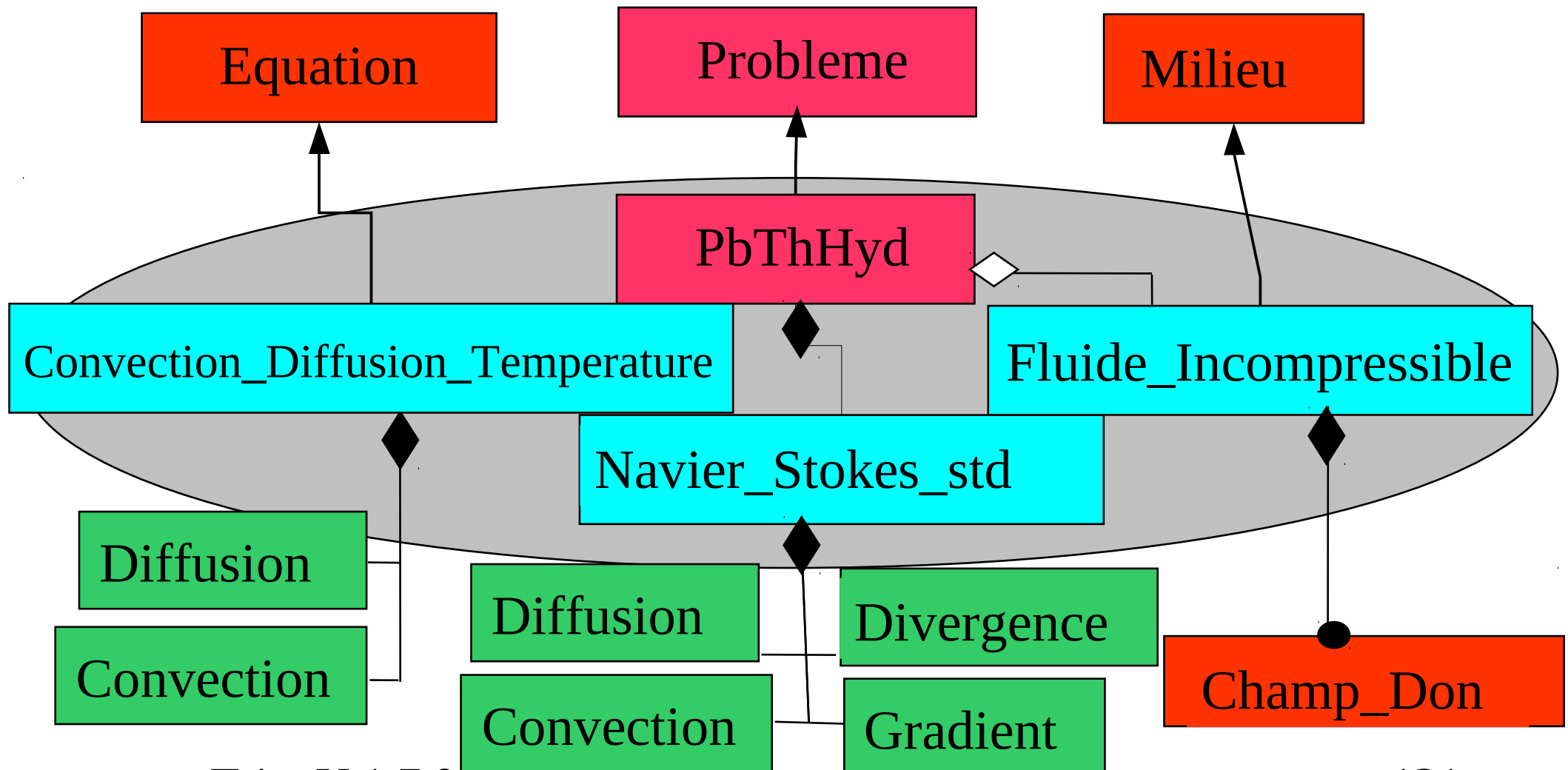
# Thermalhydraulic



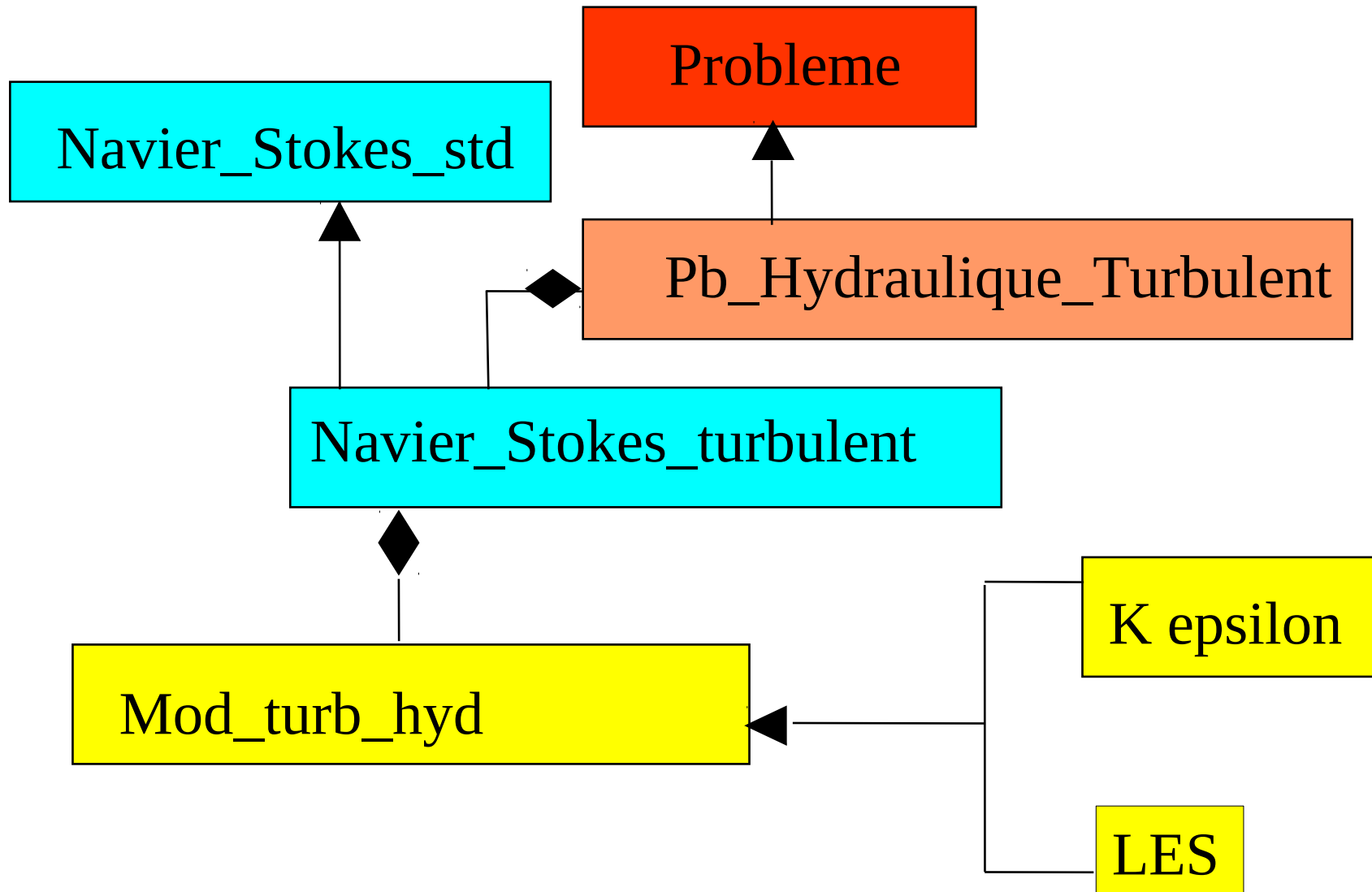
# Problem hierarchy



# Thermalhydraulic



# Thermohydraulic



# Trio\_U spatial discretization modules

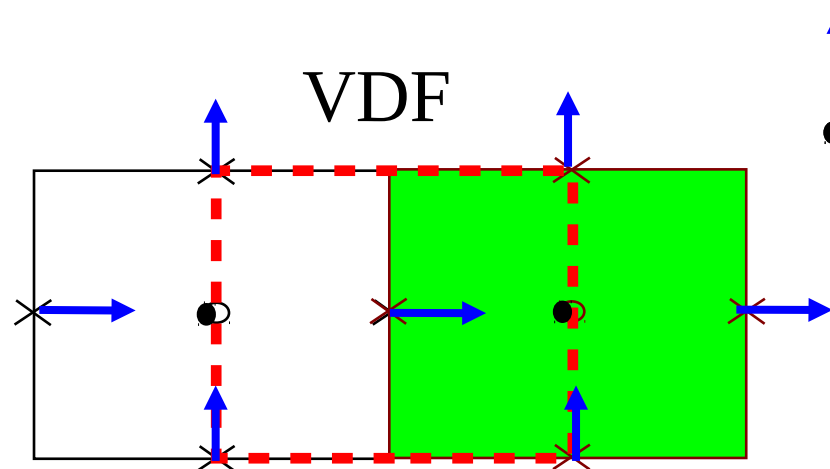
## VDF: Finite-volume differences method

More details in CHATELAIN A. thesis: <http://www.theses.fr/2004INPG0065>



## VEF: Finite-volume elements method

More details in FORTIN T. thesis: <http://www.theses.fr/2006PA066526>

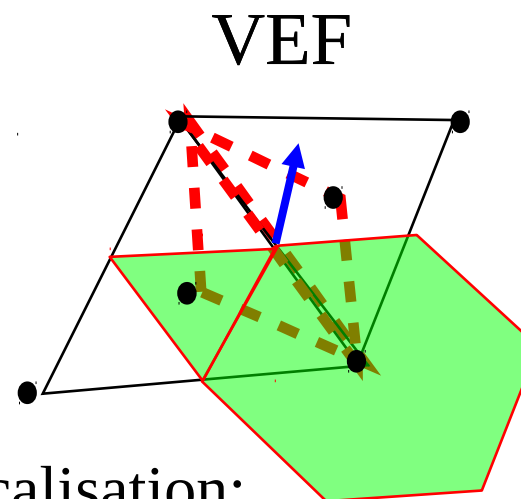
# Available discretizations




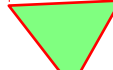
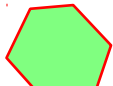
## Field localisation:

- Vector field (P1NC) at the center of the faces  
control volume: 
- Scalar field (P0) at the center of elements  
mass control volume: 

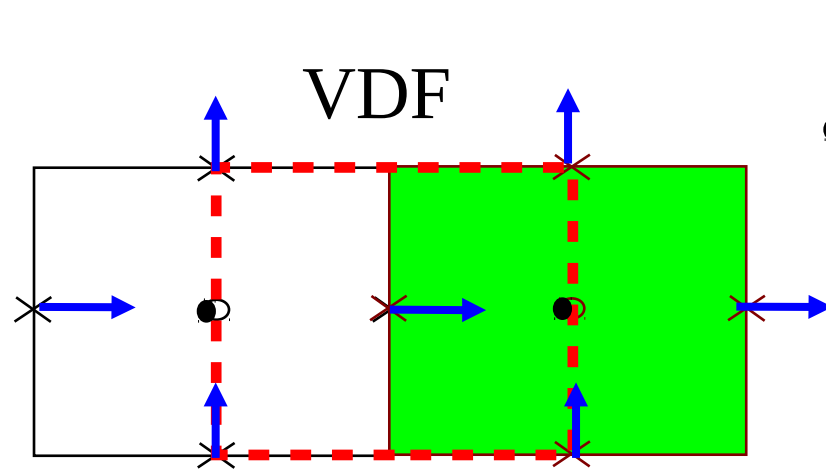
↑ Velocity  
• Pressure



## Field localisation:

- Vector and scalar fields (P1NC) at the center of the faces  
control volume: 
- Pressure (P0P1Bulle) at the nodes and the center of elements  
mass control volumes:  

# Available discretizations



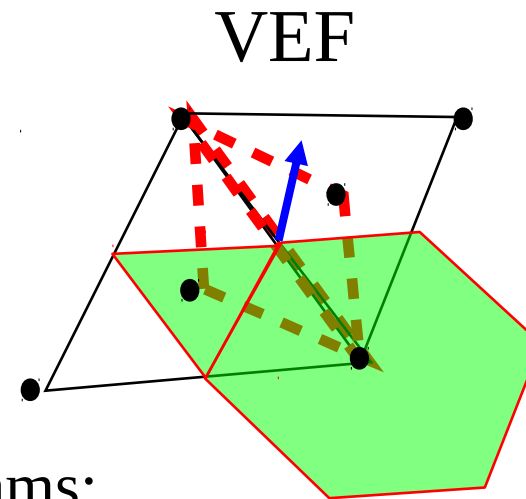
## Algorithms:

- Iterators to loop on elements or faces
- Evaluators to calculate fluxes on faces or facets

VDF/Operateurs/Iterateurs

VDF/Operateurs/Evaluateurs

↑ Velocity  
• Pressure



## Algorithms:

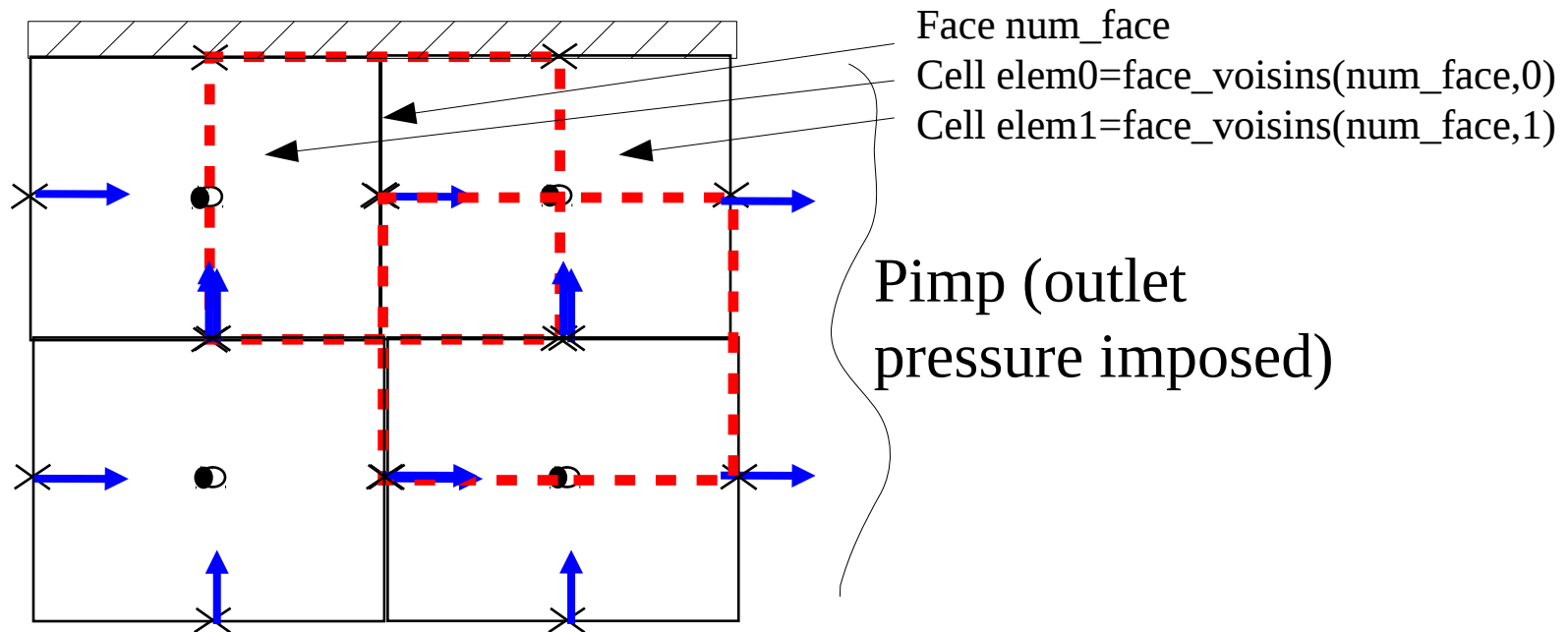
- Repeated loops on elements, faces or facets to calculate fluxes on the control volumes for each scheme

--- Momentum control volume  
■ Mass control volume

# Gradient operator example in VDF

To evaluate the volume control integration of the gradient (eg : pressure) :

$$\text{On } X \text{ axis, } \iiint \nabla P dV = \iint P.ndS = (P(\text{elem1}) - P(\text{elem0})) * \text{area}(\text{num}_{\text{face}})$$





# Gradient operator example in VDF

See `Op_Grad_VDF_Face::ajouter(const DoubleTab& inco, DoubleTab& resu)`

1) Loop on the boundaries :

**nb\_front\_cl()** returns the number of boundaries

**les\_conditions\_limites(i)** returns the boundary condition on the *ith* boundary

**face\_voisins(face,0:1)** returns the two elements surrounding the face

**face\_surfaces(face)** returns the area of the face

**bord.num\_premiere\_face()** returns the first face of the boundary *bord*

**bord.nb\_faces()** returns the number of faces of the boundary *bord*

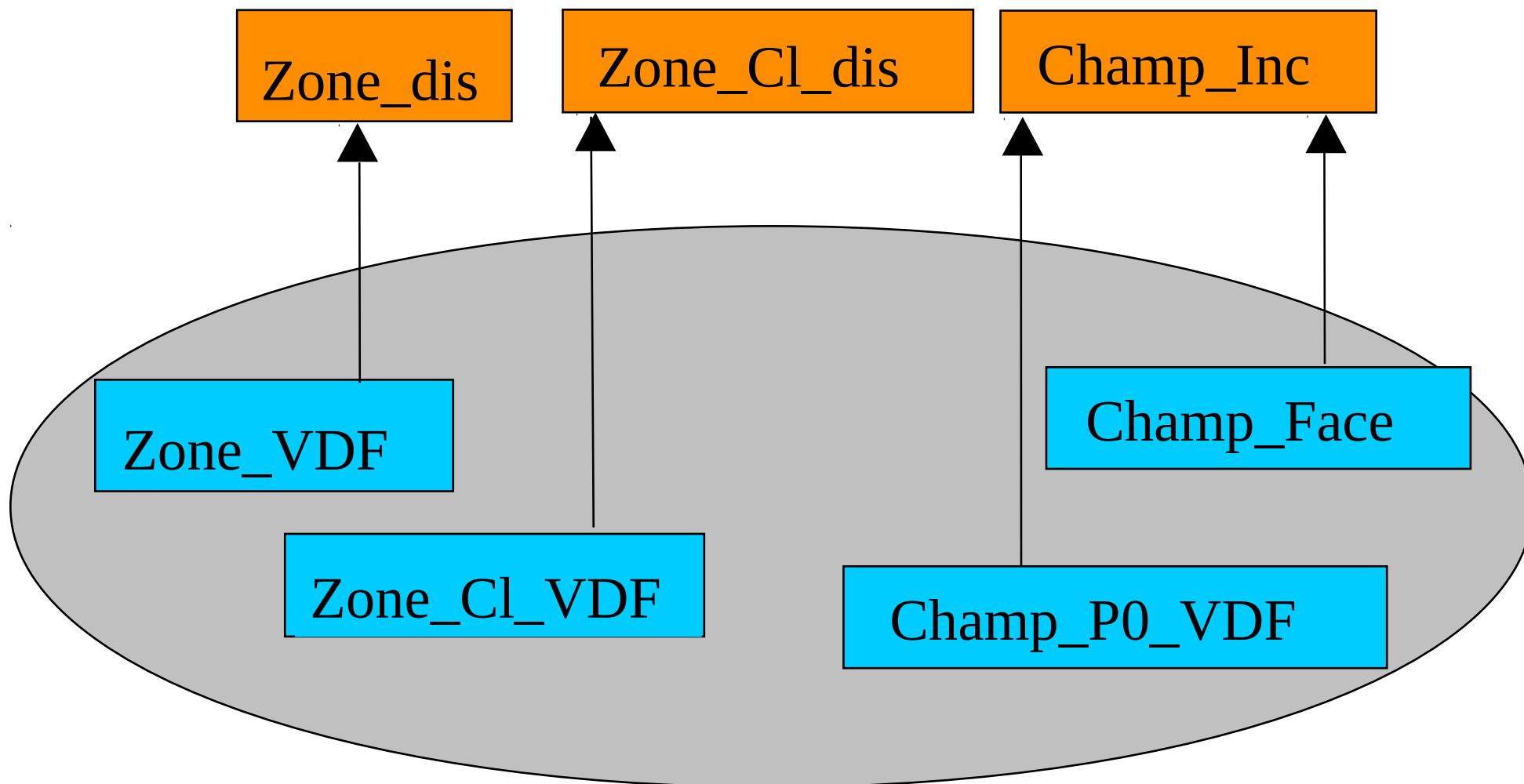
2) Loop on the internal faces :

**premiere\_face\_int()** returns the first internal face of the zone

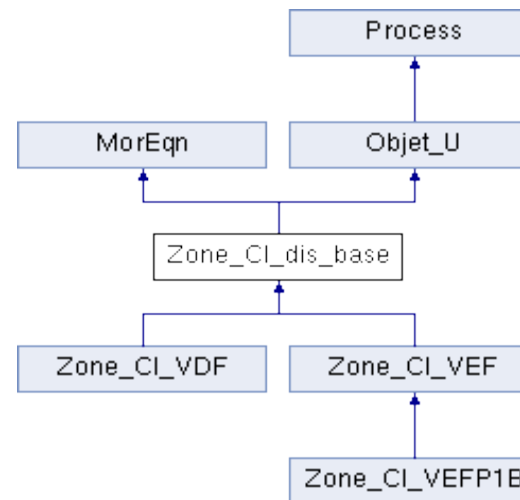
**nb\_faces()** returns the number of faces of the zone

**Remember** : Boundary faces are ranked first then internal faces in the zone.

# VDF Zones and Fields



# Zone\_Cl\_dis\_base

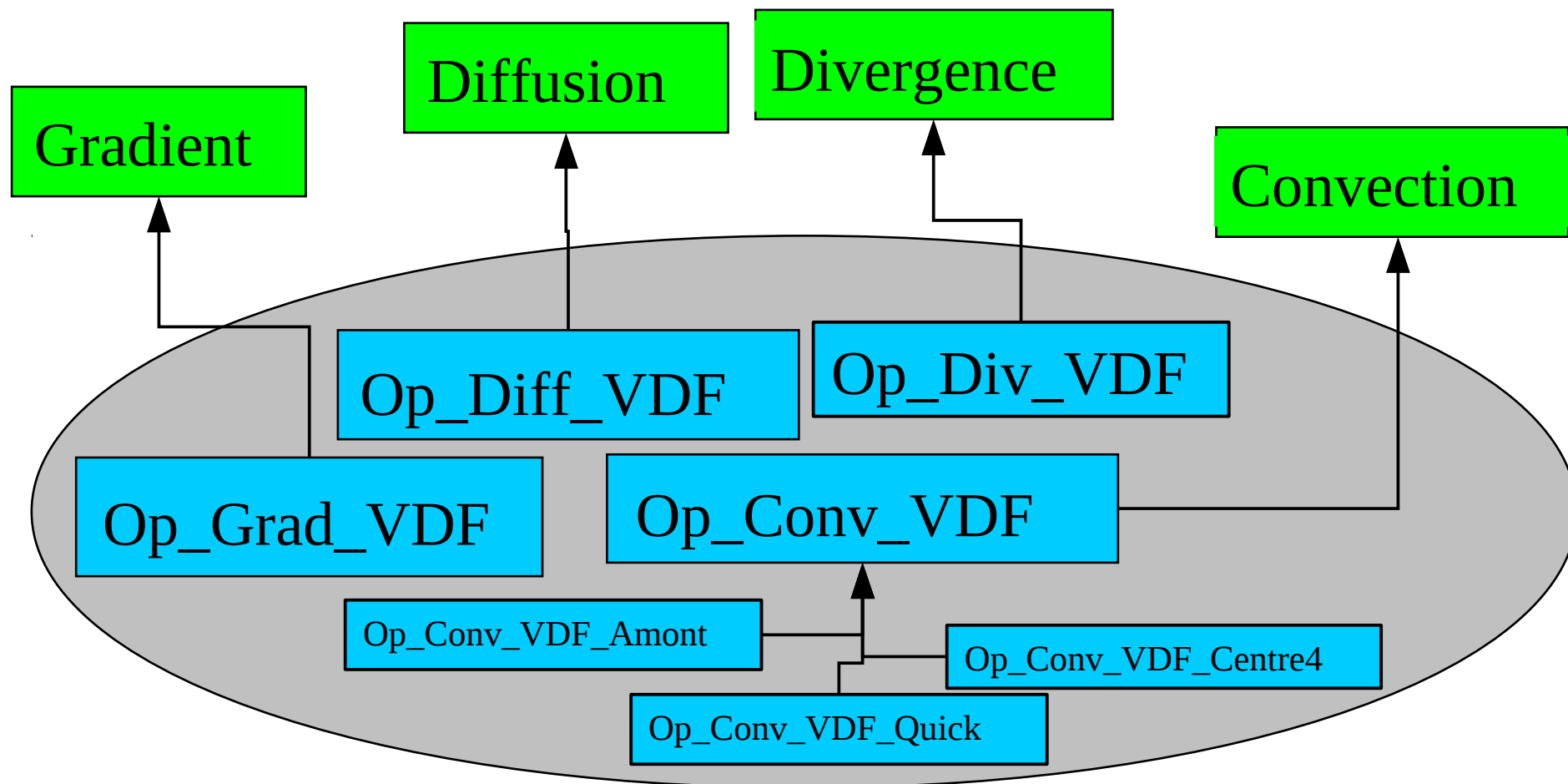


The `Zone_Cl_dis_base` classe describes discretized boundary conditions :

Protected :

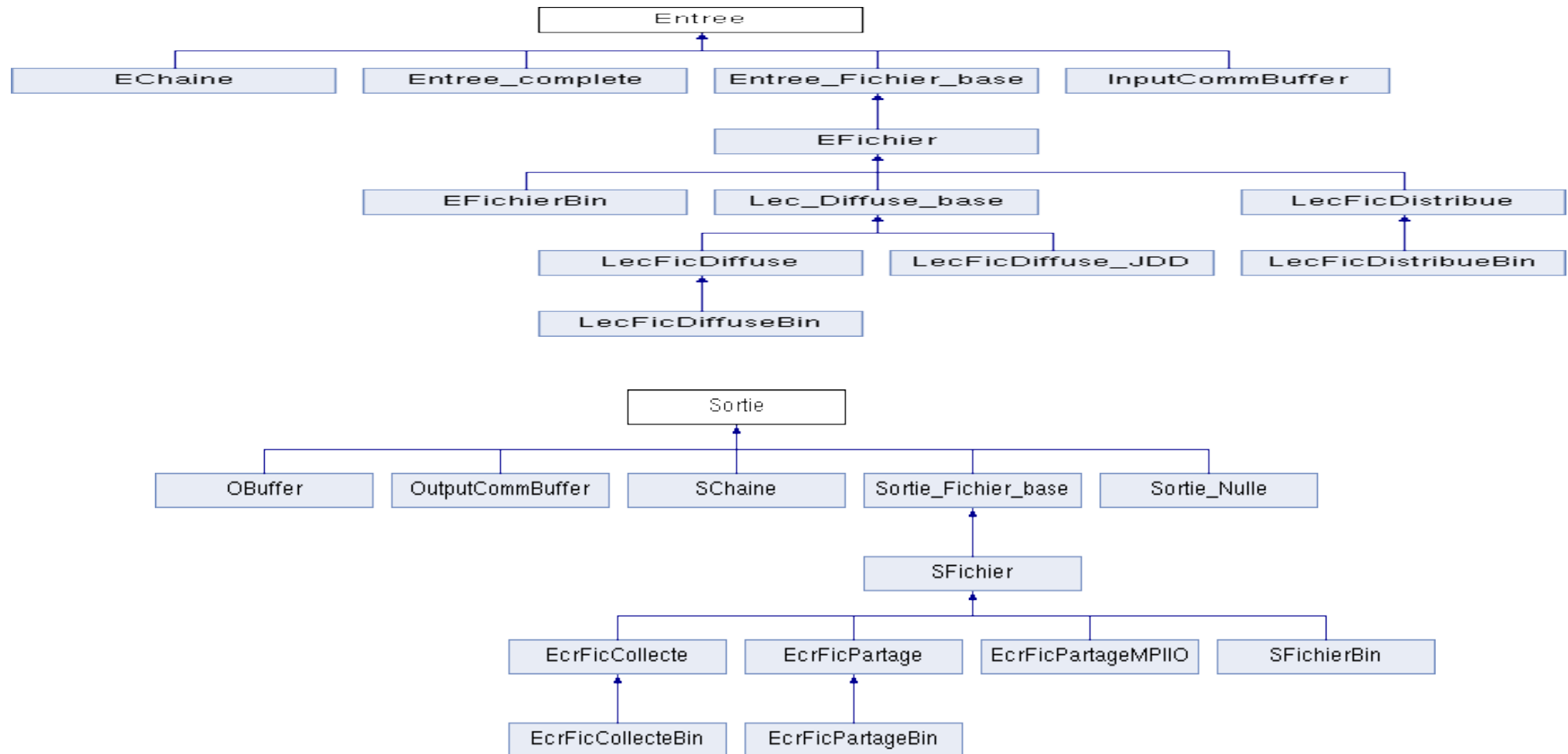
**Conds\_lim** les\_conditions\_limitees\_ ;

# Operators VDF implementation



# Managing input/output files with Trio\_U classes

# Dedicated classes to input/output



# Dedicated classes to output

**EcrFicCollecte** file(« file.txt ») ; // Each process will write in a specific file

```
file << Process::me() ;
```

**EcrFicPartage** file(« file.txt ») ; // Each process will write in the same file but sequentially

```
file << Process::me() ;
```

```
file.syncfile() ;
```

**SFichier** file(« file.txt ») ; // Each process open the same file

```
file<<Process::me() ;
```

```
// Better to use on the master process only :
```

```
if (Process ::je_suis_maitre()) {
```

```
    Sfichier file(« file.txt ») ;
```

```
    file << « Flow mass rate : »<< flow << finl ;
```

```
}
```

file\_0000.txt : 0

file\_0001.txt : 1

...

file\_000N.txt : N

file.txt : 0 1 2 3 4 ... N

file.txt : Inpredictable !

# Dedicated classes to input

```
LecFicDistribue file(« file.txt ») ; // Each process will read in a specific file_000i.txt  
file >> value ;
```

```
LecFicPartage file(« file.txt ») ; // Each process will read in the same file but sequentially  
file >> value ;
```

```
EFichier file(« file.txt ») ; // Each process will read the same file  
file>>value;  
// In this case, better to use (cause opening the same file by a lot of process is not efficient) :
```

```
LecFicDiffuse file(« file.txt ») ; // Only the master process read the file and send to other processes :  
file>>value;
```



# Baltik exercise

# Now, we are going to try to calculate the sum of the VEF control volumes on the domain in our project.

# The information is in the Zone\_VF class (a Zone\_dis discretized zone) which can't be accessed from the domain, only from the problem. So we need to read another parameter in our data file:

```
my_first_class dom { option 0 problem pb }
```

# Add the read of a new parameter problem (see [Extraire\\_plan::interpreter\\_\(Entree&\)](#) method for instance) into the *my\_first\_class.cpp* file.

# Then, remember the equation UML diagram page [123](#).

# Look for help inside the [Zone\\_VF](#), [Probleme\\_base](#) and [Equation\\_base](#) into the HTML documentation to access to the:

-equation (**equation(int )** method)

-discretized zone (**zone\_dis(int)** method)

-control volumes (**volumes\_entrelaces()** method)



# Baltik exercise

# You will need to cast the discretized zone returned by the **zone\_dis()** method into a **Zone\_VF** object.

# You will print the size of the control volumes array with something like:

```
Cerr << control_volumes.size() << finl;
```

# Where **control\_volumes** is a **DoubleVect** returned by the **Zone\_VF::volumes\_intrelaces()** method.

# If you look at the previous Problem UML diagram, you will notice a better path to access to the discretized zone: . What is this path ?

# Baltik exercise

# Now, compute and print the sum of the control volumes into a file whose name is something like:

*DataFileName\_result.txt* where DataFileName is the name of the data file (eg: Cx).

# For that, you will create the previous filename with the class **Nom** by adding to the name of the data file (given by **Objet\_U::nom\_du\_cas()** method) the string “\_result.txt” thanks to the operator+= method of the class **Nom**.

# Then you will create the file with the **SFichier** class and print the sum into this file.

# Once everything is implemented, run the test case (but, first add the keyword **FIN** just after the line where **my\_first\_class** is used in order to not run the whole calculation...)

```
$ cd ~/test/my_project/tests/Reference/NonRegression/Cx
```

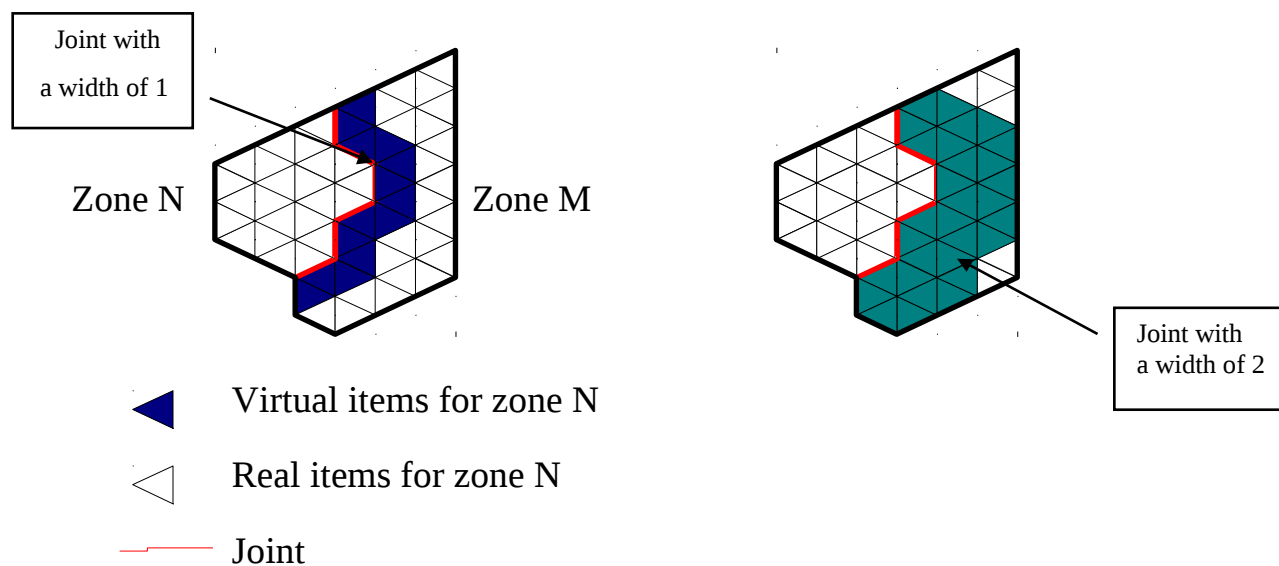
```
$ ~/test/my_project/basic Cx
```

# How to parallelize in Trio\_U

# Parallelism

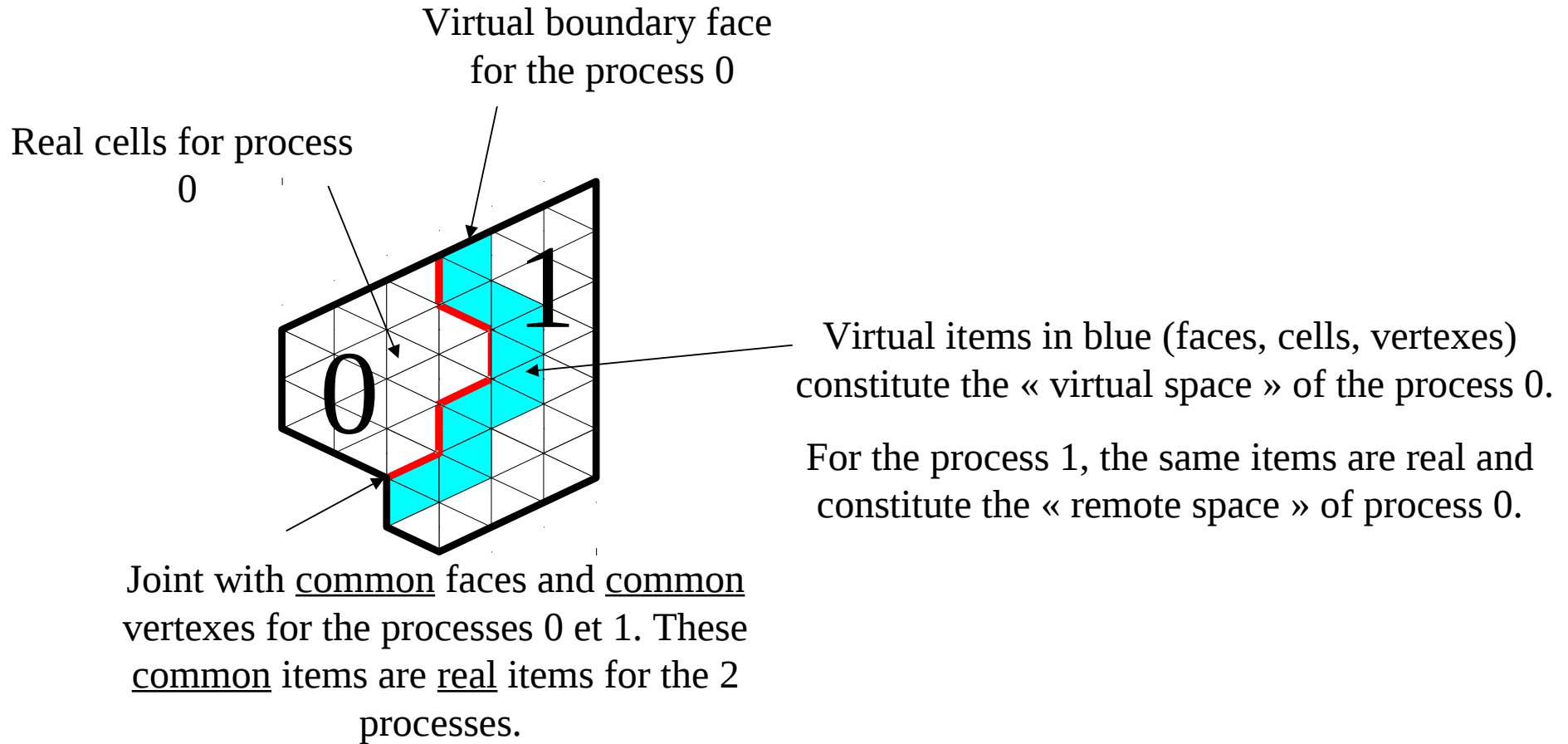
- SPMD (Single Program, Multiple Data)
- Definitions of the Trio\_U parallelism :
  - Domain partition create several Zones
  - Each process works on one Zone
  - Joint (faces that connect different Zones)
  - Items (which constitute a Zone)
    - cell, vertex, face, edge (3D)
    - may be real (physically located on the Zone) or virtual (located on the remote Zone, but known by the local process)

# Parallelism



The virtual items of the local Zone are the remote items constituted of vertexes located up to  $n$  vertexes of the  $n$ -width joint.

# Parallelism



# Parallelism

- Number of **real** items:

Zone\_VF::nb\_faces()

Domaine::nb\_som()

Zone::nb\_elem()

- Number of real+**virtual** items:

Zone\_VF::nb\_faces\_tot()

Domaine::nb\_som\_tot()

Zone::nb\_elem\_tot()

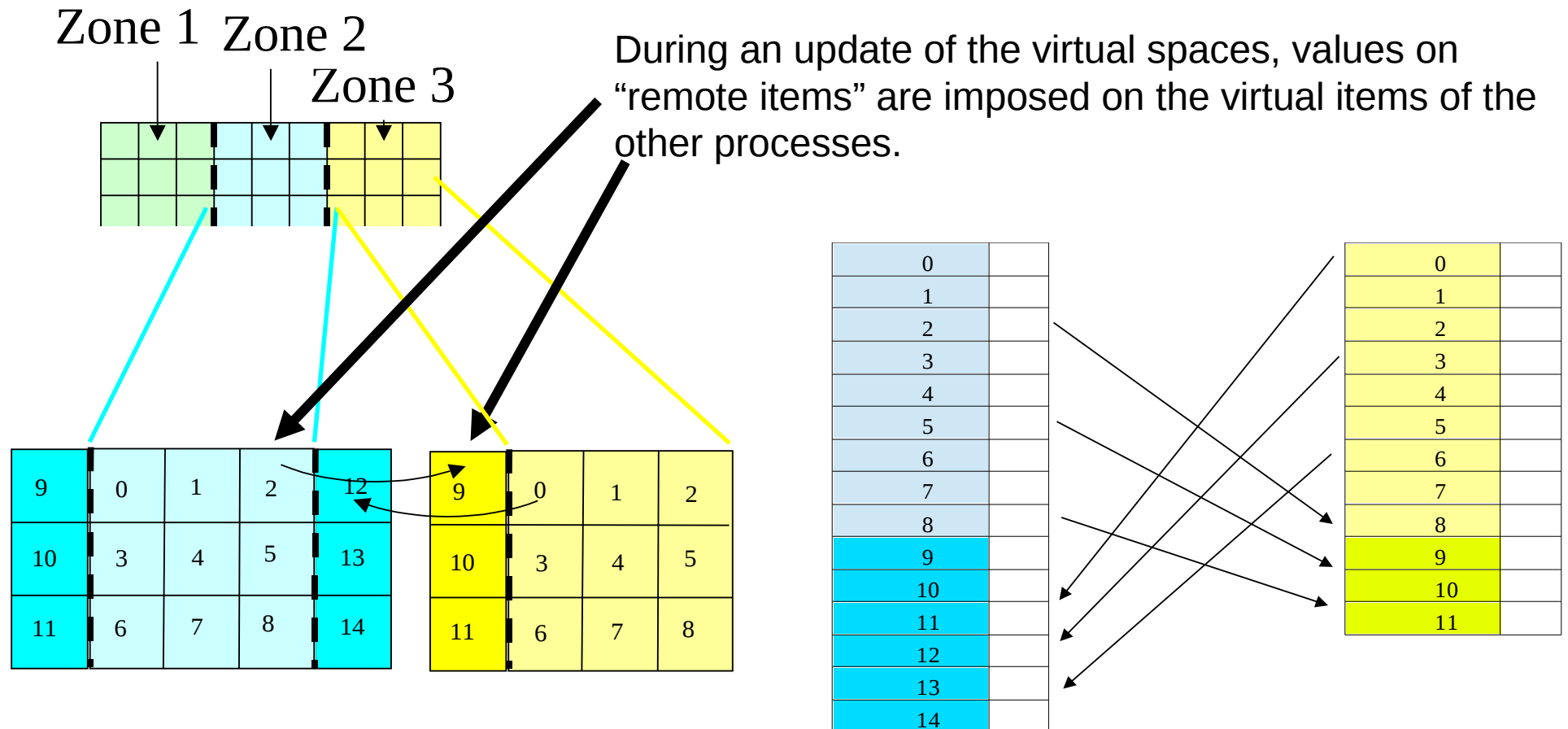
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	

Example of distributed array with additionnal data stucture (**MD\_Vector** in Trio\_U)



# Parallelism

## Example of a distributed array on cells



- Example to create a distributed array :

```
#include <MD_Vector_tools.h>
```

```
...
```

```
int nb_elem=la_zone_vef.nb_elem();
```

```
int nb_elem_tot=la_zone_vef.nb_elem_tot();
```

```
const Domaine& dom=la_zone_vef.domaine();
```

```
DoubleVect A(nb_elem);
```

```
const MD_Vector& md = la_zone_vef.zone().md_vector_elements();
```

```
MD_Vector_tools::creer_tableau_distribue(md, A); /* A has now nb_elem_tot values */
```

```
DoubleVect A(B) ; /* Or use an existing distributed array, here B */
```

```
DoubleVect C(nb_elem_tot) ; /* Warning, C is NOT a distributed array : */
```

# Parallelism

- Sizes before and after the creation of a distributed array :

```
DoubleVect A(nb_elem);
```

```
// Before :
```

```
Cerr << A.size() << finl ;      // nb_elem
```

```
Cerr << A.size_array() << finl ; // nb_elem
```

```
Cerr << A.size_reelle() << finl ; // nb_elem
```

```
Cerr << A.size_totale() << finl ; // nb_elem
```

```
const MD_Vector& md = domaine().zone().md_vector_elements();
```

```
MD_Vector_tools::creer_tableau_distribue(md,A);
```

```
// After :
```

```
Cerr << A.size() << finl ;      // nb_elem
```

```
Cerr << A.size_array() << finl ; // nb_elem_tot
```

```
Cerr << A.size_reelle() << finl ; // nb_elem
```

```
Cerr << A.size_totale() << finl ; // nb_elem_tot
```

# Parallelism

- Update of the virtual space of a distributed array is done by:  
`tableau.echange_espace_virtuel();`
- Notes:
  - `echange_espace_virtuel()` does **nothing** on real arrays
  - It is possible to check if an update of the virtual space is useful or not with :  
`#include <Check_espace_virtuel.h>`  
`....`  
*`/* Exit in error if the virtual spaces of the distributed array A are not up to date */`*  
`assert(check_espace_virtuel_vect(A));`

# Parallelism

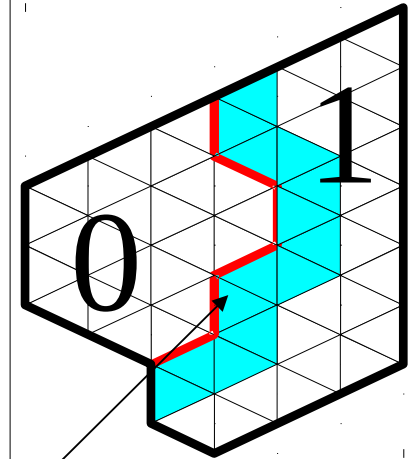
## **When do I need to create a distributed array ?**

- It depends of your algorithm and the items you are using
- Use carefully distributed arrays. It will slow down the parallel execution during each virtual spaces update
- Example where you need it: You want to calculate the interpolation of a cell centered field to the faces of the mesh :



# Parallelism

```
// Non distributed array of a cell centered field :
const entier nb_elem=zone_VEF.nb_elem() ;
DoubleVect Field(nb_elem) ;
// Loop on cells to fill the array Field :
....
// Now to calculate the faces interpolation of this field
const entier nb_faces=zone_VEF.nb_faces();
DoubleVect A(nb_faces);
// Loop on the real faces and use Zone_VF ::face_voisins() distributed array
...
// Problem : values on joint common faces are not well evaluated
// cause there is no virtual space on Field array to access virtual cells, so the
// good solution would be to create a distributed version for Field :
MD_Vector_tools::creer_tableau_distribue(md, Field);
// Loop on real cells to fill the array Field
...
Field.echange_espace_virtuel() ; // To update the virtual spaces of Field array
// Loop on real faces to fill A
```



# Parallelism

- Some useful Trio\_U methods to know from the **Process** class:
  - **Process::je\_suis\_maitre()** returns 1 if the current process is the master process 0
  - **Process::me()** returns the current number process
  - **Process::nproc()** returns the process numbers
  - **Process::mp\_sum(x)** returns the sum of x on the whole processes
  - **Process::mp\_min(x)** returns the smallest value of x
  - **Process::mp\_max(x)** returns the biggest value of x
  - **Process::barrier()** waits that all processes reach this point

# Parallelism

- On the arrays:
  - **mp\_somme\_vect**(DoubleVect& x) returns the sum of all the elements from the distributed vector x
  - **mp\_norme\_vect**(DoubleVect& x) returns the L2 norm of the distributed array vector x
  - **mp\_norme\_tab**(const DoubleTab& x, ArrOfDouble& y) returns in the array y the L2 norm of each component of the distributed array x
  - **DoubleVect::mp\_moyenne\_vect()** returns the mean of the distributed vector x
- Standard/error output:
  - Cout : only the master process writes to standard output
  - Cerr : only the master process writes to error output, but other processes write to .log files
  - Cout\_Global/Cerr\_Global : all the processes write to standard/error output
  - Journal() : all the processes write to the .log files



# Parallelism

- Send/receive methods (envoyer/recevoir). Well described in the file :
  - \$TRIO\_U\_ROOT/Kernel/Utilitaire/communications.cpp
  - Example of use in the [Sous\\_Zone.cpp](#) file. An array is sent by the master processor (0) and received by all the other ones.

# Parallelism

## – Pitfall with the common items :

*/\* During the sum of the values of a vertex located array tab, the following loop is incomplete : \*/*

```
double sum=0 ;
```

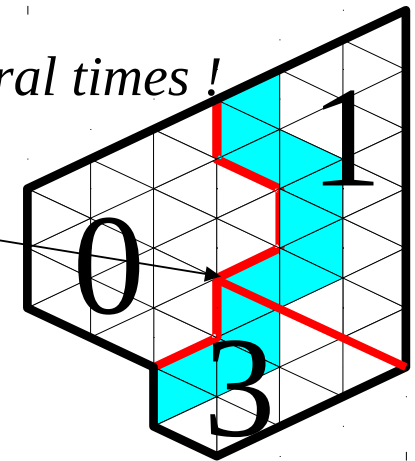
```
for (int i=0;i<nb_som;i++)
```

```
    sum+=tab(i);
```

```
sum=Process::mp_sum(sum);
```

*// Cause the common vertexes are counted several times !*

Common vertex counted 3 times in the sum



**NB:** In this case, you would use :

```
double sum = mp_somme_vect(tab) ;
```

# Baltik exercise

*# Run your test case Cx in parallel mode:*

```
$ cd ~/test/my_project/tests/Reference/NonRegression/Cx
```

```
$ make_PAR.data Cx 2 # Partition in 2 subdomains
```

```
$ export exec=~/test/my_project/basic
```

```
$ triou PAR_Cx 2 # 2 processes used
```

*# Compare the files: Cx\_result.txt, PAR\_Cx\_result.txt and explain the discrepancies between the values.*

*# To parallelize the algorithm, rewrite it, according to the previous slide with the help of the **mp\_somme\_vect(DoubleVect&)** method and change the way the result is written in the .txt file. You should find the same value for the sequential and parallel calculation.*

# Parallelism

-Pitfall with how the faces are ranked in Trio\_U (Zone\_VF class) :

- First, the real boundary faces (from 0 to nb\_faces\_int()-1)
- Second, the real internal faces (from nb\_faces\_int() to nb\_faces()-1)
- Last, the virtual faces, internal or boundary with no particular order (from nb\_faces() to nb\_faces\_tot())

*// So, to loop on the internal faces, you will write :*

```
const int nint=zone_VF::premiere_face_int();
const int nb_faces_tot=zone_VF::nb_faces_tot();
for (int face=nint;face<nb_faces_tot;face++)
    if (!zone_VF.est_une_face_virt_bord(face))
        .... // Internal face (real or virtual)
```

# Parallelism

```
// Loop on the boundary faces
for (int i=0;i<les_cl.size();i++)
{
    const Cond_lim& la_cl = les_cl[i];
    const Front_VF& le_bord=ref_cast(Front_VF,la_cl.frontiere_dis());
    int nb_faces_bord_tot = le_bord.nb_faces_tot();
    // Loop on real and virtual faces of a boundary :
    for (j=0 ;j< nb_faces_bord_tot;j++)
    {
        int face=le_bord.num_face(j);
        ....
    }
}
```

**Warning:** Some obsolete code is still using the old way to access virtual faces on boundaries: Zone\_VF::ind\_faces\_virt\_bord

# Parallelism

- How to debug parallelization in Trio\_U
  - build your code in debug mode to take advantage of all the implemented checks (asserts) in the code
  - test your parallelization :
    - on several test cases with different meshes
    - vary the partition number N of the different meshes
    - the explicit parallel run command is :  
**mpirun -np N \$exec\_debug datafile N**
  - What if the parallel calculation crashes/hangs ?
    - Give a try with the debugger to know exactly where the issue is :  
**mpirun -gdb -np N \$exec\_debug datafile N**

# Parallelism

## – How to validate parallelization in Trio\_U

Check the results are the same on  $N=1$  and  $N>1$  cpus :

- Create a reference with a sequential calculation (post process some fields at LATA format):

**triou datafile.data**

- Run you parallel calculation on  $N$  cpus and compare the LATA results :

**triou parallel\_datafile.data  $N$**

**compare\_lata datafile.lata parallel\_datafile.lata**

- The **compare\_lata** tool will compare all the post-processed fields in the two files and will warn if the relative differences are bigger than  $1.e-5$ , which may indicate an incorrect parallelization

# Parallelism

How to find the source(s) of parallelism differences in Trio\_U ?

-Use the **Debug** keyword by inserting in the sequential and parallel data files after the **Discretize** keyword:

**Debug** problem\_name seq faces 1.e-6 0 # In the sequential datafile

**Debug** problem\_name seq faces 1.e-6 1 # In the parallel datafile

-Run the sequential then the parallel calculation. The **Debug** keyword will compare arrays each time this line is found in the code :

```
Debug::verifier(« I am checking array », array);
```

-Look at the log files to detect when the parallel difference appears.



# Parallelism

## How to validate performance improvements

- Run sequential and parallel calculations on clusters with an optimized version of the code
- Look the CPU measures into the files :
  - datafile.TU # *Contains the global performances*
  - datafile\_detail.TU # *Contains the per process performances*

### Statistiques d'initialisation du calcul

Temps total 2.99584

### Statistiques de resolution du probleme

Temps total 3.46542

Timesteps 3

Secondes / pas de temps 1.14932

Dont solveurs Ax=B 0.805794 70% (1 appel/pas de temps)

Dont operateurs convection 0.157865 13% (2 appels/pas de temps)

Dont operateurs diffusion 0.053469 4% (2 appels/pas de temps)

Dont operateurs gradient 0.02917 2% (2 appels/pas de temps)

Dont operateurs divergence 0.00428367 0% (2 appels/pas de temps)

Dont operateurs source 0.01545 1% (1 appel/pas de temps)

Dont operations postraitements 0.0103403 0% (1 appel/pas de temps)

Dont calcul dt 0.00864567 0% (4 appels/pas de temps)

Dont modele turbulence 0.0473803 4% (1 appel/pas de temps)

Dont calcul divers 0.0169207 1%

Nb echange\_espace\_virtuel / pas de temps 404.333

Nb solveur / pas de temps 1

Secondes / solveur 0.805794

Iterations / solveur 126.667

Communications avg 17.7 % of total time

Communications max 21.4 % of total time

Communications min 14 % of total time

Network latency benchmark 7.10487e-07 s

Network bandwidth max 236.697 MB/s

Total network traffic 66.9368 MB / timestep

Average message size 41.0824 kB

Min waiting time 1.7 % of total time

Max waiting time 9.1 % of total time

Avg waiting time 5.4 % of total time

# Trio\_U test coverage

# Code coverage

- Created by gcov tool, as a nightly task on ~2000 test cases.
- 65% of Trio\_U total lines are covered (Cerr & exit lines excluded)
- Knowing the coverage of methods/functions of the code gives confidence (or not) when re-using it for your development.
- Trio\_U code coverage and tools exploiting it are available for the developer

# Useful code coverage tools

## **triou -check class::method**

-Trio\_U tool to know and run the test cases covering a method.

-For Baltik developer : Not available yet ?

### ***Example :***

```
$ triou -check Navier_Stokes_std::mettre_a_jour
```

**triou -check all|testcase:** Option to check the non-regression on one or several test cases

Example :

```
$ triou -check VAHL_DAVIS
```

For Baltik developer (version=optim|debug):

```
$ make check_version # Check the project non-regression on Baltik test cases
```

```
$ make check_trio_version # Check the project non-regression on Trio_U test cases
```

# Code coverage exercice

# Browse the Trio\_U ressources index file :  
**triou -index**

# Select the Test coverage link :

Q: Which is the less covered matrix class ?

Q: Run the test cases using the RRK2 time scheme.

# Trio\_U coding rules

# Coding rules

- Class name = File name
- One class per file
- Respect modularity :
  - Kernel should be built without VDF or VEF module
  - VDF application should be built without VEF module
  - ...
- Use assert() for pre and post conditions when coding a method
- Use Param object to read keyword parameters
- ...

# Coding rules

- Do not use pointers but instead the classes :
  - REF for association
  - DERIV for generic class
  - VECT/LIST
- Use Kernel arrays (Double|IntVect...)
- No french accents
- Cerr/Cout in english in Kernel module
- ...



# Rules to contribute

You want your work to be merged in the next release of the Trio\_U/Kernel,  
then provide to the Trio\_U support team :

## I) If you develop directly in Trio\_U/Kernel :

- English description/syntax of the new keywords
- The name of the branch under Git you have pushed to the shared repository containing :
  - New/modified sources
  - New validation forms or test cases
- Non regression should have been checked (no errors) on the debug binary and possible differences should be explained. Run :

**exec=\$exec\_debug triou -check all**

# Rules to contribute

You want your work to be merged in the next release of the Trio\_U/Kernel,  
then provide to the Trio\_U support team :

## II) If you develop in a Baltik project based on Trio\_U/Kernel :

- English description/syntax of the new keywords
- If not using Git, provide a tar.gz package containing your work (new/modified sources, validation forms/test cases,...) with :
  - make distrib
- Non regression should have been checked (no errors) on the debug binary and possible differences should be explained :
  - make check\_all\_debug # Check non regression of the Baltik and Trio\_U/Kernel
  - VALGRIND=1 make check\_all\_optim # Same in optimized mode with Valgrind check

# Using Eclipse

# IDE Eclipse

Download : <http://www.eclipse.org>

Check you have also :

Egit (Git support) : <http://www.eclipse.org/egit>

Cdt (C++ support) : <http://www.eclipse.org/cdt>

-> Create a new C/C++ project (Makefile Project with Existing Code) under  
`$TRIO_U_ROOT/src`

We are only in an evaluation phase...

# After the training session...

Read the commented solution of the exercise :

`$TRIO_U_ROOT/doc/Trio_U/exercices/my_first_class`

Practice on a tutorial :

`$TRIO_U_ROOT/doc/Trio_U/exercices/equation_convection_diffusion`

# The End

Good luck!  
[triou@cea.fr](mailto:triou@cea.fr)