

---

# TRUST 1.7.4

## developer's training session

---

# Table of contents

- Introduction..... p3
- TRUST an object oriented CFD code..... p7
- TRUST specifications/choices explained..... p12
- TRUST/TrioCFD modules..... p15
- Using **Eclipse**..... p22
- **BALTIK: Building an Application Linked with Trio\_U Kernel**..... p24
- Basic OOC concepts used in TRUST..... p28
- The extensive use of **macros** in TRUST..... p48
- Four different kind of classes in TRUST (**Base, Instanciate, Associated & Generic classes**).... p54
- Interpretors: Links between data file and the code (Eg: **Read & Solve**)..... p73
- Exploring the **Kernel (Math & Framework), ThHyd & space discretization** modules..... p82
- Managing **input/output** files with TRUST classes..... p122
- How to **parallelize** in TRUST ..... p126
- TRUST **test coverage** ..... p148
- How to **debug** TRUST..... p152
- TRUST **coding rules** ..... p158

# Introduction

# Prerequis

For this training session:

- TRUST/TrioCFD (User's training session)
- C++ (Intermediate)

Later, if you want to develop/contribute to TRUST:

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



# Objectives

To get a general knowledge of the TRUST code

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

To acquire reflexes to develop while following TRUST rules of coding



# Useful links

## TRUST:

<http://sourceforge.net/projects/trust-platform/files/>

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

<mailto:triu@cea.fr>

## C++:

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

## Git:

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

<http://www.alexgirard.com/git-book/index.html>

# TRUST

## An object oriented CFD code

# Interest of TRUST

- 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 TRUST support team



# Interest of TRUST

- 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 TRUST)

# What is TRUST CFD code ?

## It provides :

- 3 spatial discretizations (VDF, VEF, EF)
- 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

# TRUST

- What can handle TRUST
  - Runs on every Linux box (32/64 bits)
  - Runs on the CEA clusters
    - Has already run a LES on a  $400.10^6$  cells mesh with 10000 cores (curie on CCRT)

# TRUST

## Specifications/Choices explained

# Main specifications:

Enable developments with the following characteristics:

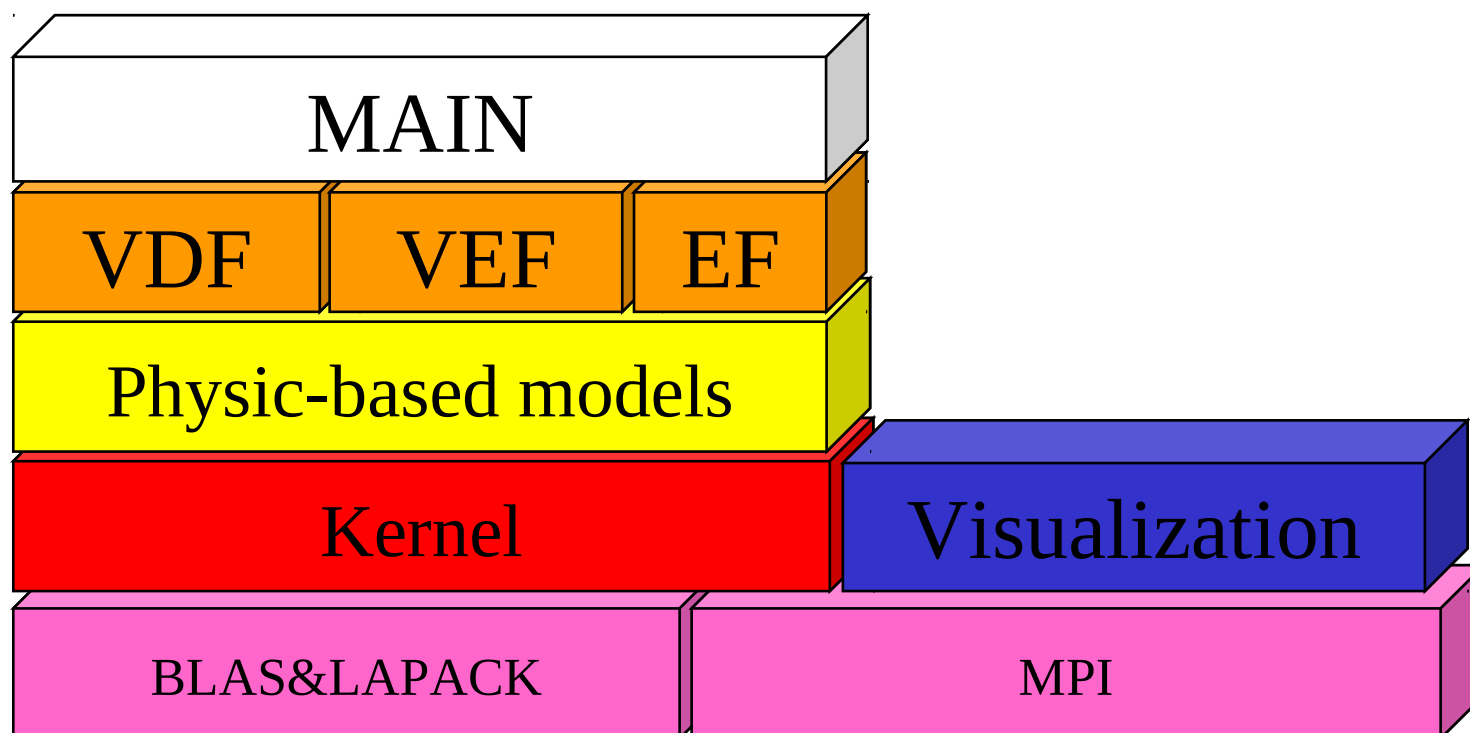
- fast
- reliable
- reusable
- effective
- documented
- enable encapsulation of Fortran modules

# 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 (**Git**)
  - Co-developing
- Automatic generation via Doxygen of **HTML** documentation from code sources
  - Documentation is up to date

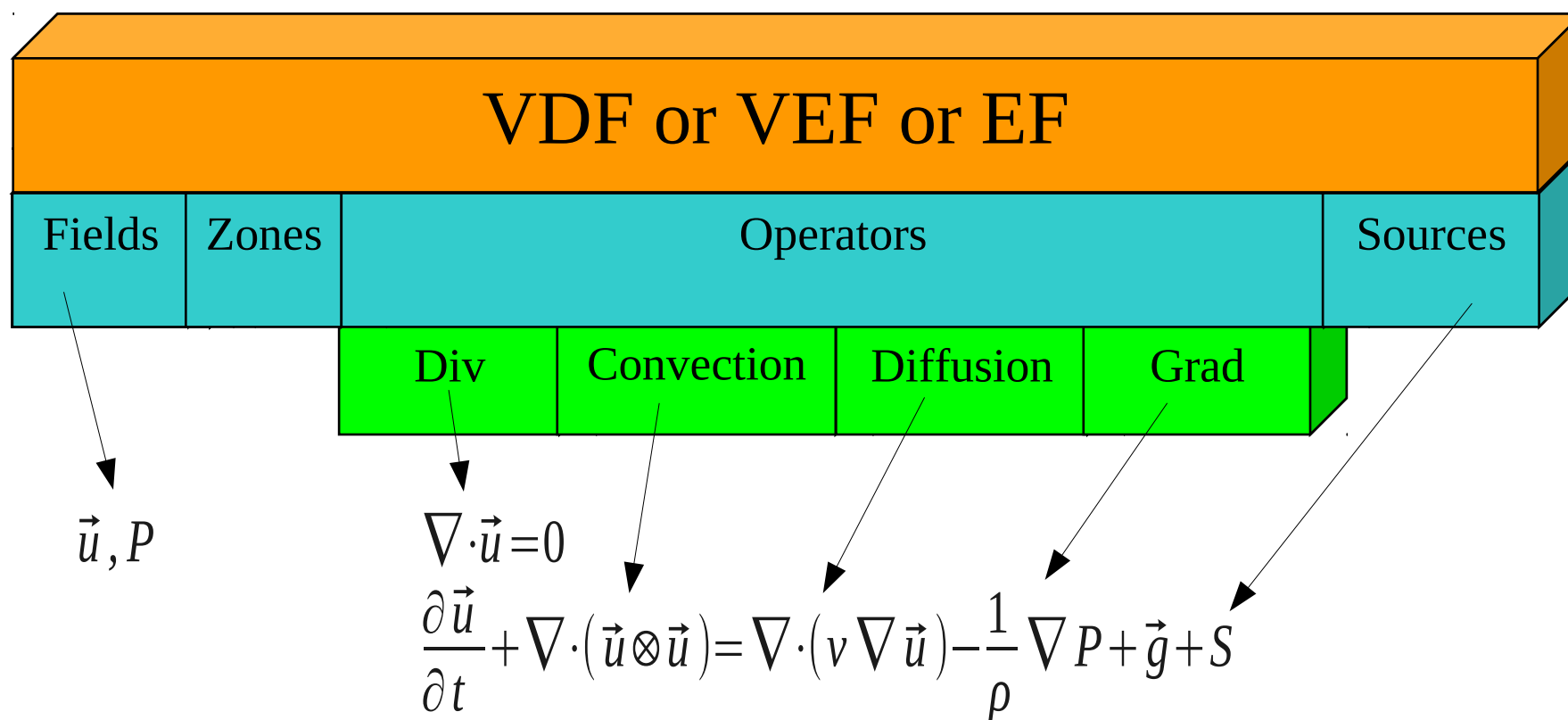
# TRUST modules

# TRUST modules

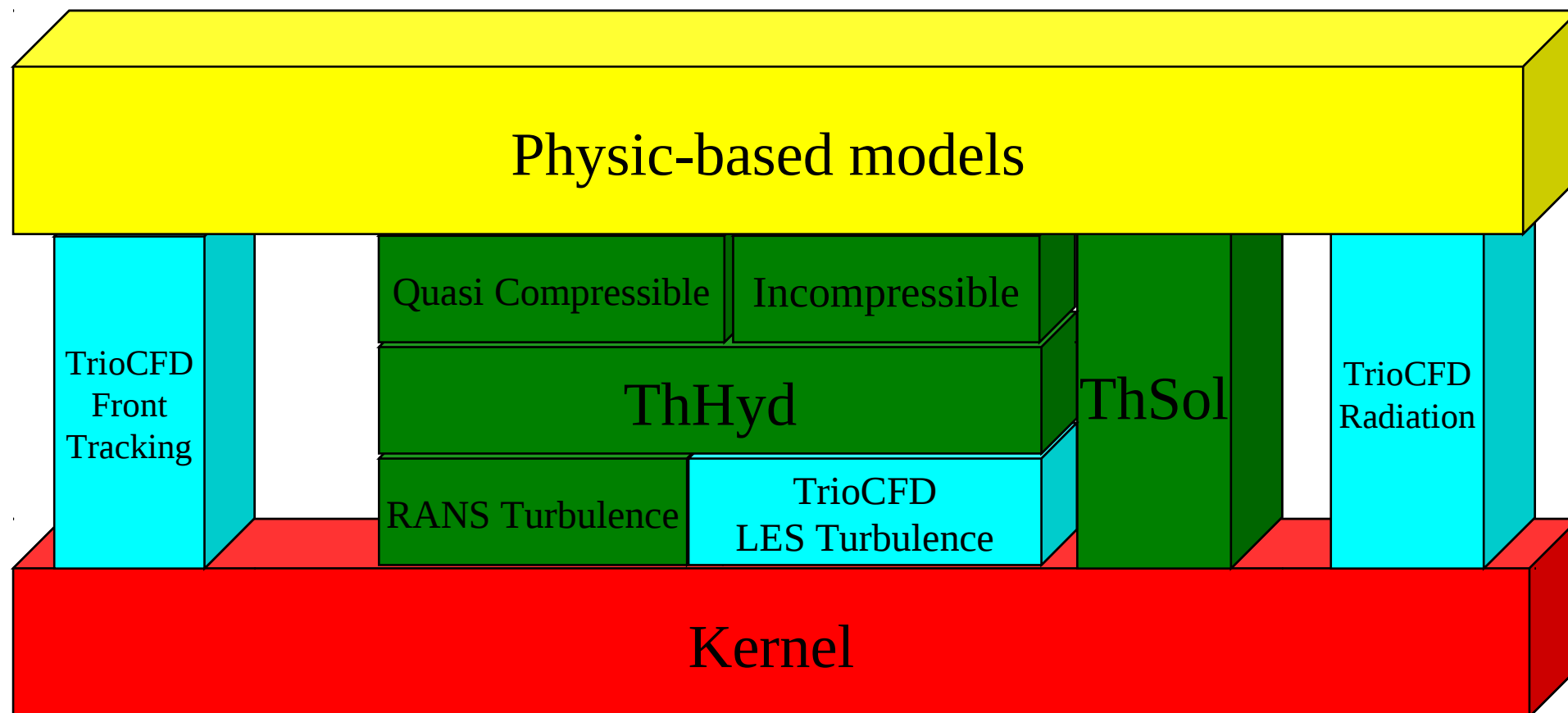




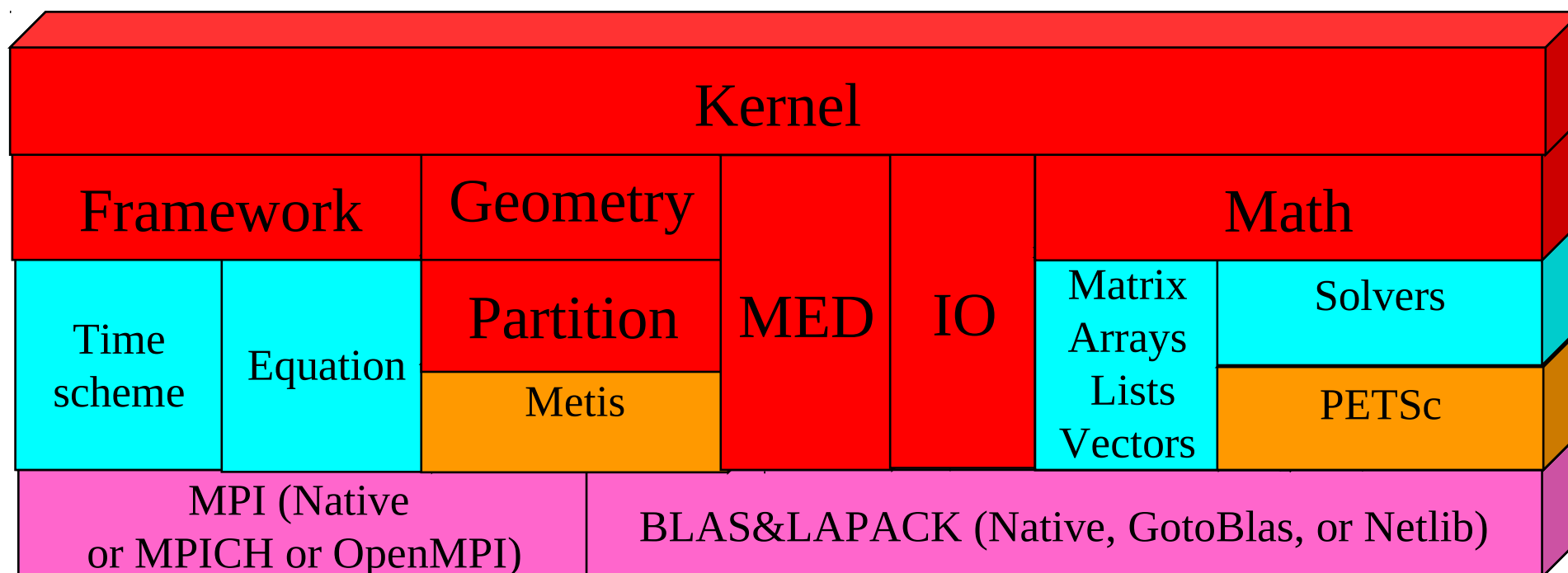
# Discretization modules



# Physics modules



# Kernel module



# TRUST sources

- TRUST code is made of:
  - 1600 classes
  - Declared in include files (.h)
  - Implemented in sources files (.cpp)
  - Within 74 directories
  
- Kernel constitutes 47% of the TRUST code.
  
- There is a HTML documentation to browse and see the class hierarchy under:  
[\\$TRUST\\_ROOT/doc/html](#)  
Or  
[trust -index](#)

# TRUST tests

~60 **TRUST Verification forms** to check analytical results under:

**[\\$TRUST\\_ROOT/Validation](#)**

~150 **TrioCFD Validation forms** to compare with experimental results or with results from other codes under:

**[\\$project\\_directory/validation](#)**

~2050 **Non regression test cases**:

~750 TRUST non regression test cases under **[\\$TRUST\\_ROOT/tests](#)**

~ 1300 TrioCFD non regression test cases under **[\\$project\\_directory/build/tests](#)**

# Using Eclipse

# TRUST Baltik project Tutorial

## - Load the TRUST environment:

```
source /home/triou/env_TRUST_X.Y.Z.sh
```

## - Open the TRUST tutorial:

```
trust -index
```

→ « Tutorial » link in the developer block

## → TRUST Initialization exercise

Main page: <http://www.eclipse.org>

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

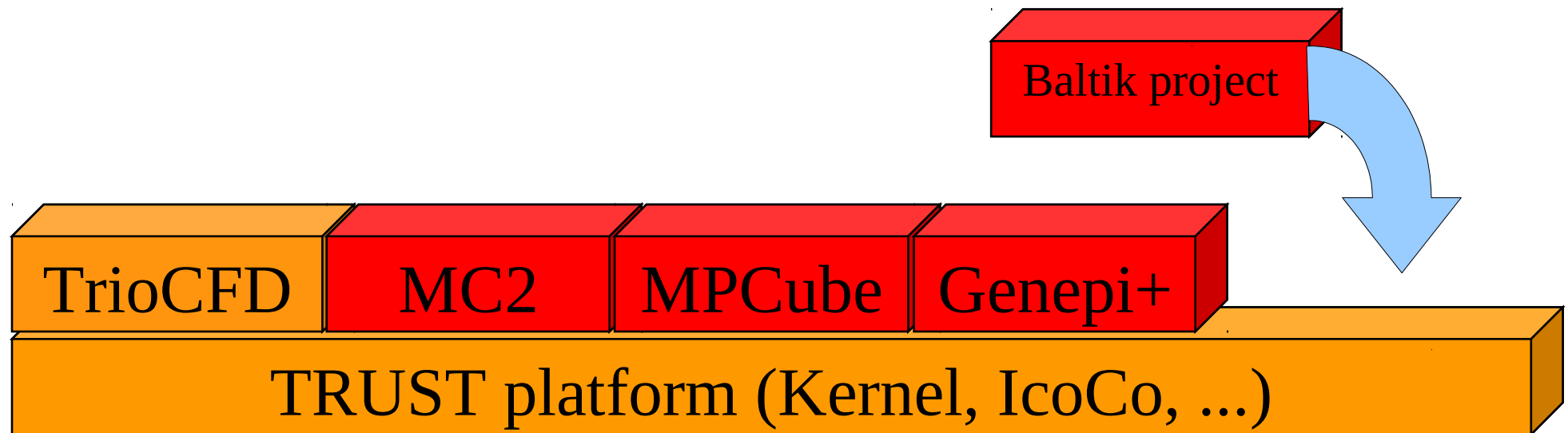
Cdt (C++ support): <http://www.eclipse.org/cdt>

# Baltik

## **B**uilding **A**pplication **L**inked with **T**rio\_ **U** **K**ernel



# Develop in a TRUST Baltik project



I) with new features

II) modifying TRUST functions

You need to first load TRUST environment.

# Develop in a TRUST Baltik project

## I) Develop in a Baltik project based on TRUST

- You want to develop your own project
  - more freedom about the update of TRUST version
- Baltik means **B**uilding an **A**pplication **L**inked to **T**rio\_ **U** **K**ernel

## II) Integrate your project in TRUST base

- You want to contribute to TRUST
- But if you want to share your work, you will need :
  - To follow the TRUST roles of coding
  - To check and respect the non regression of others parts of the code
  - To add new validation forms or test cases

# TRUST Baltik project Tutorial

## → Baltik initialization exercise

- Creation of a Baltik project
- Creation of your git repository
- Builds
- Using Eclipse

# Basic Oriented Object Conception (OOC) concepts used in TRUST

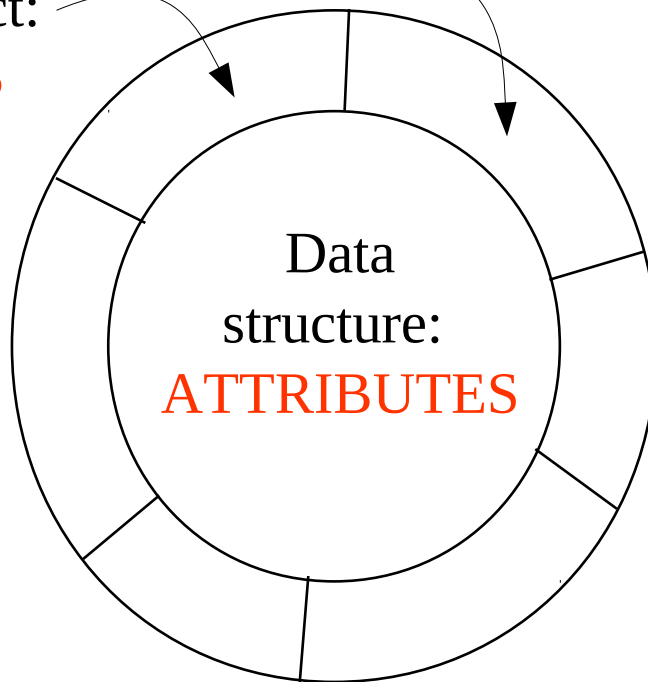
# What are C++ class/object?

- 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 TRUST 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))
```

```
...
```

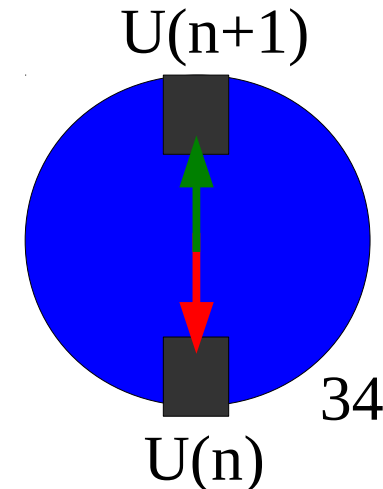
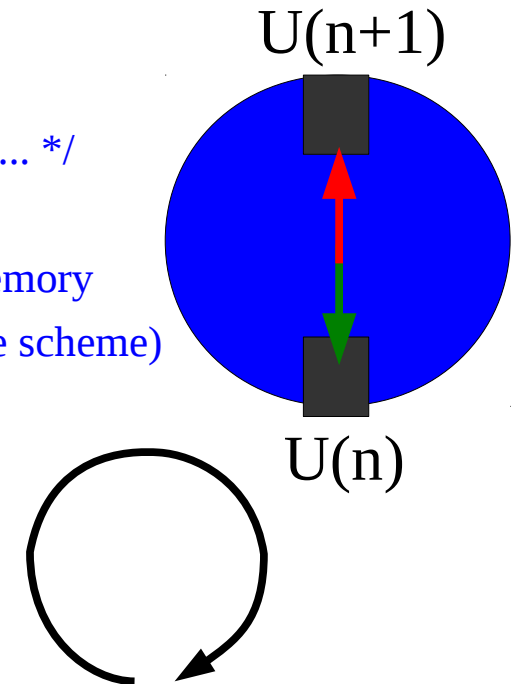
```
// At the end of the time step, we turn the « wheel » with:
```

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

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

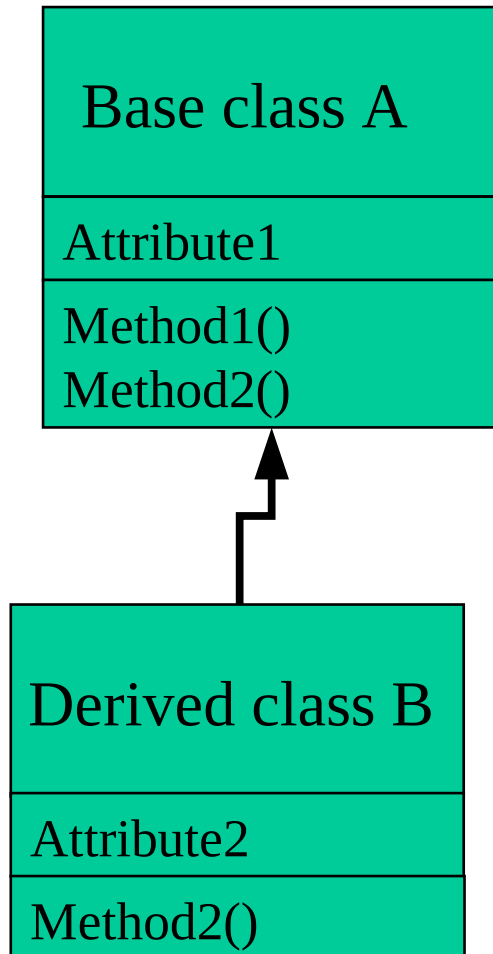
```
// So during, the next time step, the memory used to store U(n) (now useless)
```

```
// 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:

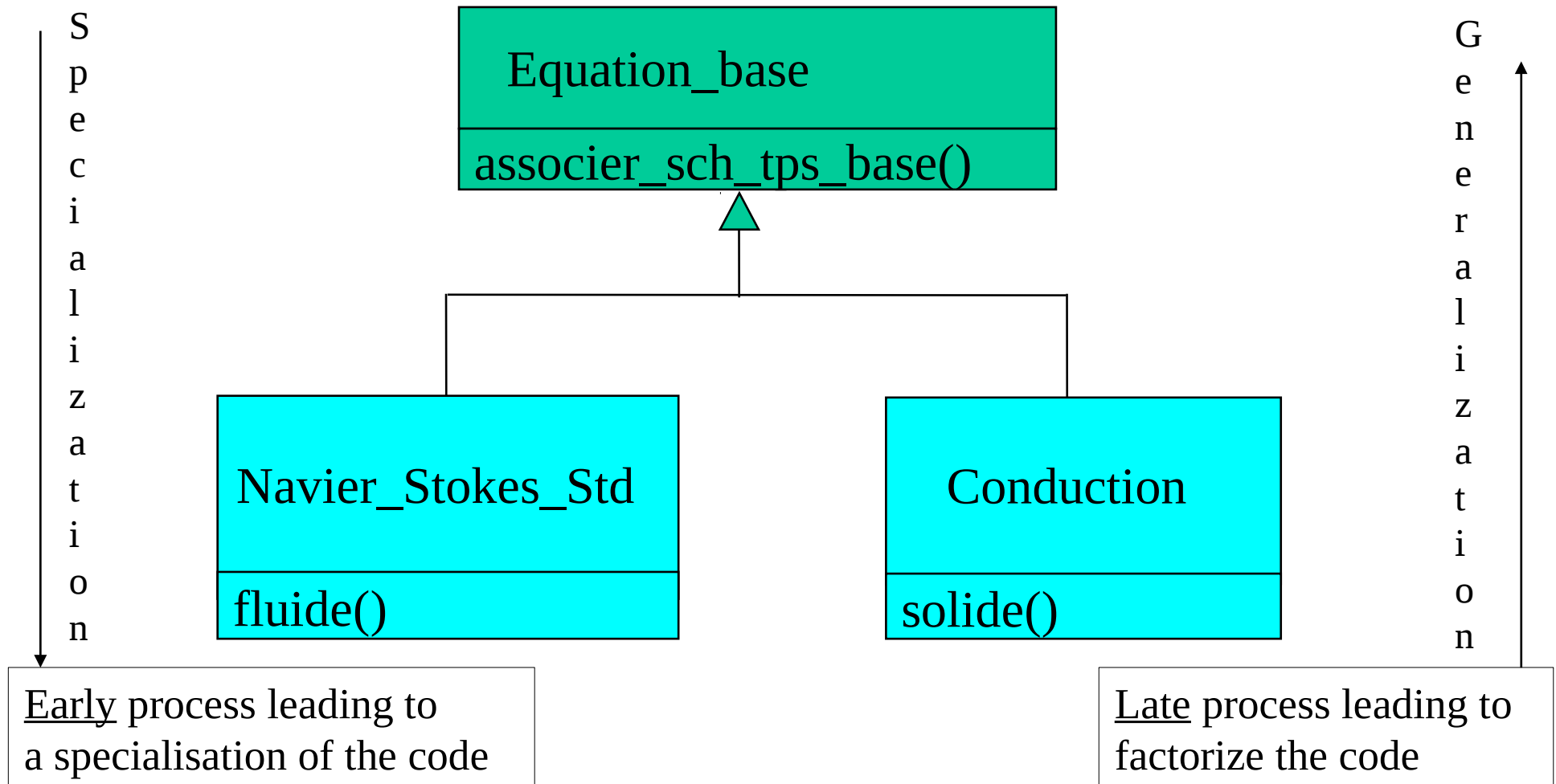
-> Attribute1 and Method1() are **inherited** from the class A

-> 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



# Polymorphism use in TRUST

→ 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 TRUST  
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

**TRUST 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):

$$1) BU = 0$$

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

Or by inverting 2) by M gives 2'):

$$\begin{aligned} 2') \partial U / \partial t &= -M^{-1} B^T P + M^{-1} (LU - CU + S) \\ &= -\mathbf{M^{-1} B^T P} + F(U) \end{aligned}$$

Then using 1) on 2') leads to 1'):

$$1') BM^{-1} B^T P = BM^{-1} (LU - CU + S) \quad \Rightarrow \mathbf{P^{n+1}}$$

Solving 2'):

$$2') \partial U / \partial t = -\mathbf{M^{-1} B^T P} + F(U) \quad \Rightarrow \mathbf{U^{n+1}}$$

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

# Virtual method example

derivee\_en\_temps\_inco() is a virtual method of Equation\_base class, who calculates  $\partial U / \partial t = F(U) = M^{-1}(\sum Op_i(U) + \sum S_i)$ :

```
DoubleTab& Equation_base::derivee_en_temps_inco(DoubleTab& F)
{
    // for explicit case
    F=0;
    DoubleTrav secmem(F);           // Initialisation by copy
    // Loop on the operators to add them to the second member of the equation
    for(int i=0; i<nombre_d_operateurs(); i++)
        operateur(i).ajouter(secmem);    //  $\sum Op_i(U)$ 
    // Adding source terms
    les_sources.ajouter(secmem);        //  $\sum Op_i(U) + \sum S_i$ 
    // Call to an other virtual method
    corriger_derivee_expl(secmem);       // do nothing except for Navier_Stokes_std (overloaded): returns -GradP
    solveur_masse.appliquer(secmem);    // ->  $M^{-1}(\sum Op_i(U) + \sum S_i)$ , and for Navier_Stokes_std:  $M^{-1}(\sum Op_i(U) + \sum S_i - \text{GradP})$ 
    F=secmem;
    F.echange_espace_virtuel();         // parallel instruction
    corriger_derivee_impl(F);           // for Navier_Stokes_std: calculates  $P^{n+1}$ , also used by Transport_K_Eps
    return F;
}
```

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

# Virtual method example

```
DoubleTab& Navier_Stokes_std::derivee_en_temps_inco (DoubleTab& F)
{
    // for explicite case
    //  $M \partial U / \partial t + \text{grad } P = MF(U) = \Sigma Op_i(U) + \Sigma S_j$ 
    //  $\text{div } U = 0 \rightarrow \text{div } M^{-1} \text{grad } P = \text{div } F(U)$ 
    Equation_base::derivee_en_temps_inco(F); //  $F(U) - M^{-1} \text{grad } P$ 
    return F;                               //  $\partial U / \partial t = F(U) - M^{-1} \text{grad } P$ 
}
```

Navier\_Stokes\_std::derivee\_en\_temps\_inco(F)

→ Equation\_base::derivee\_en\_temps\_inco(F)

→ corriger\_derivee\_expl(secmem) **which is overloaded in** Navier\_Stokes\_std class  
to calculate -GradP !

# 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)
{
    //  $\partial U / \partial t = F(U^n)$  -->  $U^{n+1} = U^n + dt * F(U^n)$  for forward Euler scheme
    DoubleTab& present = eqn.inconnue().valeurs(); // Contains  $U^n$ 
    DoubleTab& futur = eqn.inconnue().futur(); // Location to store  $U^{n+1}$ 
    DoubleTab dudt(futur); // Copie of  $U^{n+1}$ 
    // Using boundary conditions applied on  $U^{n+1}$ :
    ...
    eqn.derivee_en_temps_inco(dudt); //  $F(U^n)$ 
    ...
    //  $U^{n+1} = U^n + dt * dU/dt$ 
    futur=dudt;
    futur*=dt_; //  $dt * F(U^n)$ 
    futur+=present; //  $dt * F(U^n) + U^n$ 
    eqn.zone_Cl_dis()->imposer_cond_lim(eqn.inconnue(),temps_courant()+pas_de_temps());
    ...
    return 1;
}
```

## 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 : ...

-> ...

# TRUST Baltik project Tutorial

**→ PRM file and validation test cases  
exercise**

# The extensive use of macros in TRUST



# TRUST important points

TRUST does not use, for historical reasons:

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

-Instead of templates, TRUST uses macros

-Instead of using STL, TRUST defines LIST, VECTORS,...

# TRUST important points

No pointers in TRUST:

- to avoid coding error
- to differentiate the aggregation of the reference

You will never see:

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

But instead:

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

# TRUST important points

## Why no pointers in TRUST?

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)
}
```

# TRUST **macros**

Macros are widely used to implement plumbing of several features of TRUST. 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)

# TRUST 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)
- To ensure a correspondence dataset /class

# Four different kind of classes in TRUST:

**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.


## **TRUST 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 »
}
```



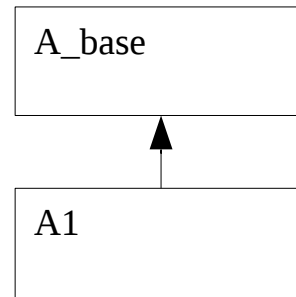
# Four different kind of classes in TRUST:

Base class  
**Instantiate class**  
Associated class  
Generic class

# 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)  
{  
    ...  
}  
...
```

# 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);
```

```
}
```

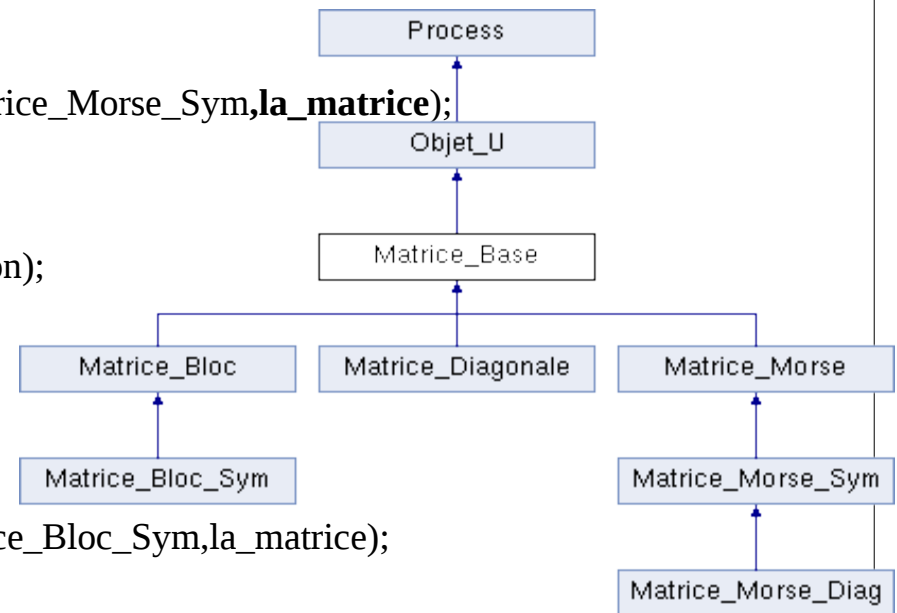
```
else if(sub_type(Matrice_Bloc_Sym,la_matrice))
```

```
{
```

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

```
    Matrice_Morse_Sym mat_sym;
```

...



# Four different kind of classes in TRUST:

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 TRUST:  
**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_;
}
```

# 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

**WARNING : use only REF(A) in your code.**



## 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);
```

# Four different kind of classes in TRUST:

Base class  
Instantiate class  
Associated class  
**Generic class**



# 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 pointed type of the object, which is different of the object type given by the **que\_suis\_je()** method. Example :

**Conduction** cond; // Instanciaded class

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

**Equation** eqn; // Generic class

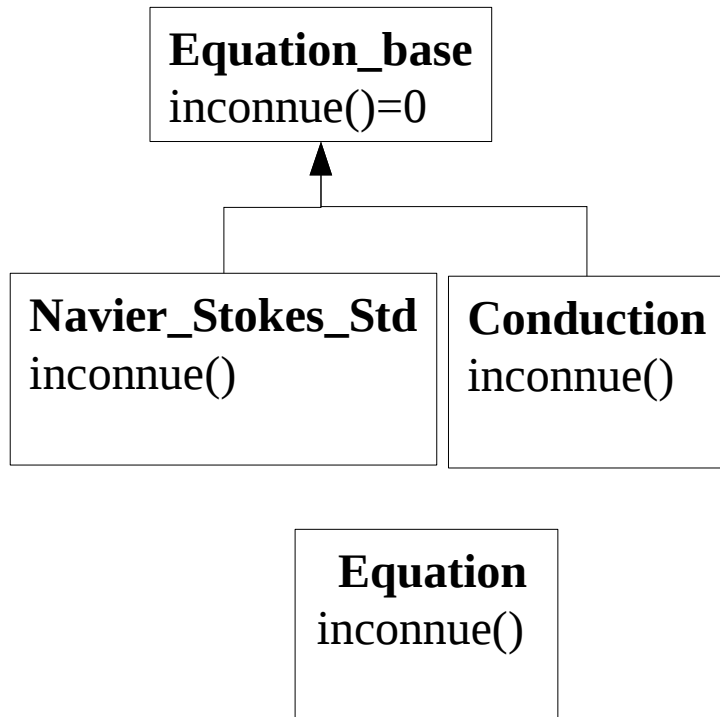
eqn.typer(Conduction) ;

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

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

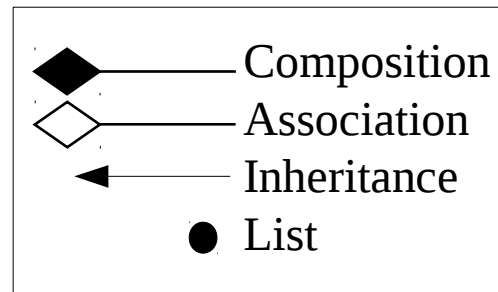
- Often (but not always), hierarchy methods are also coded in generic classes to avoid the use of **.valeur()**. Example :

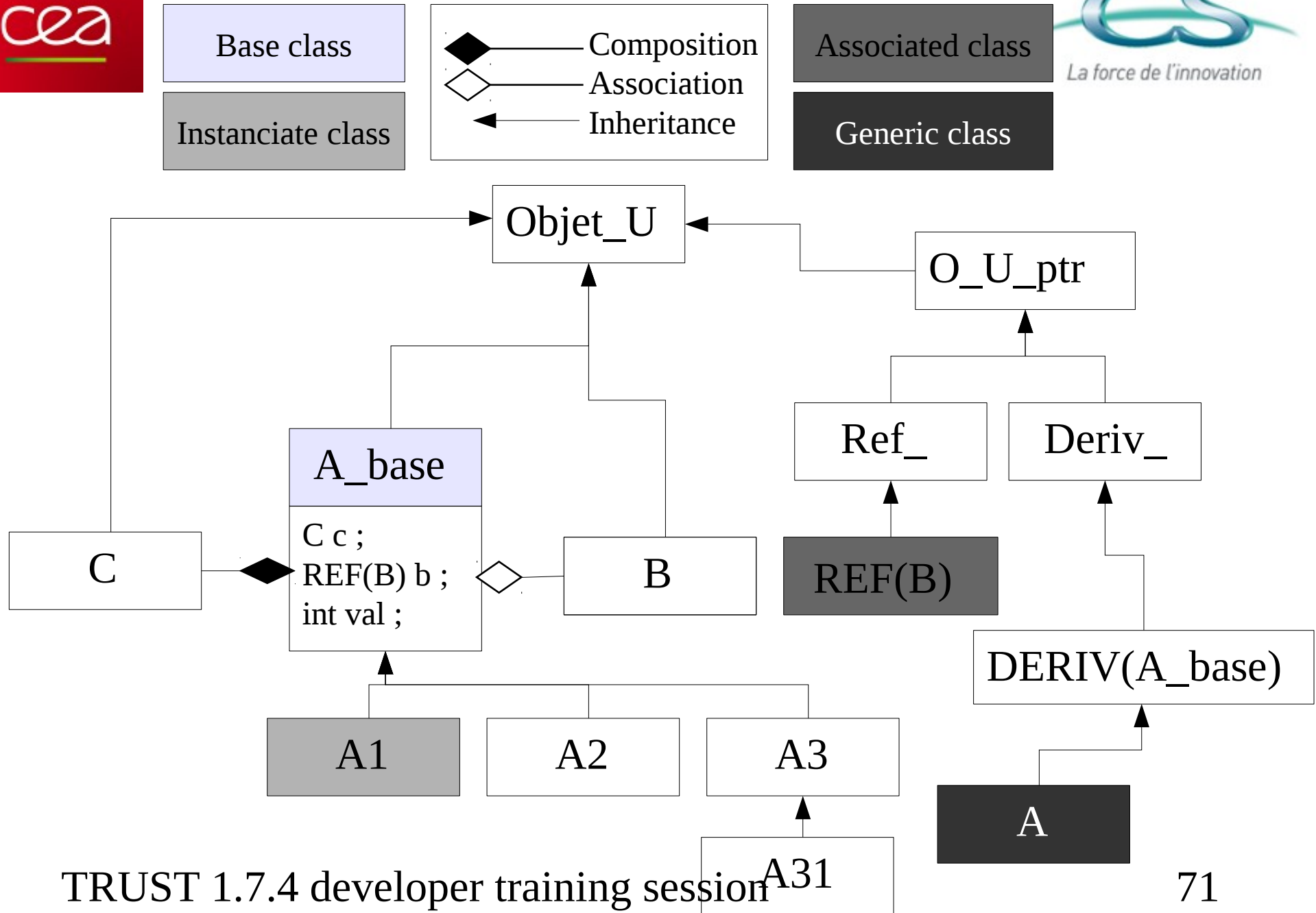
```
Champ_Inc& Equation::inconnue()
{
    return valeur().inconnue() ;
}
```



# Hierarchy examples and UML notations

UML (Unified Modeling Language)





## Exercise: Use HTML doc

# Browse the TRUST ressources index file :

\$ trust -index

# Notice when you select a class, the **localisation of his .cpp/.h files are specified at the bottom of the html page.**

# 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 ?



# Interpretors: Links between data file and the code

**Read:** keyword to read an Object  
**Solve:** keyword to solve a Problem

# Which method is called ?

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

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

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

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

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

# Interpretors: Links between data file and the code

Read: keyword to read an Object

**Solve:** keyword to solve a Problem

# Where is solved a problem ?

```
Pb_hydraulique pb
...
Read pb { ... }
Solve pb
```

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

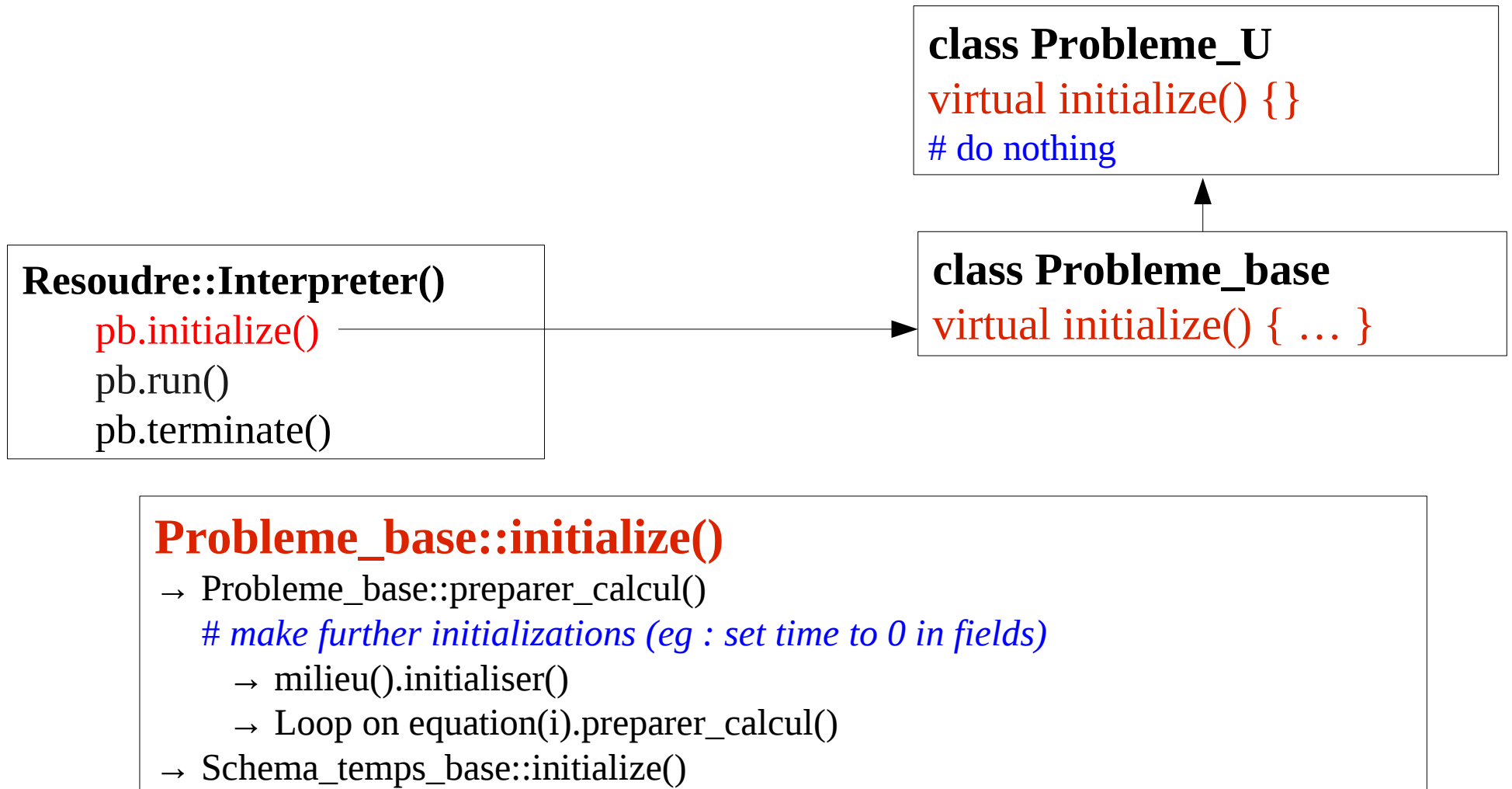
- The **Solve** interpreter solve the **problem**
- The object **problem** is described by a class which inherits from :

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



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

# Resoudre call graph





## 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() *// Update each unknown & BC & media*
  - Schema\_Temps\_base::mettre\_a\_jour() *// t(n+1)=t(n)+dt(n)*
  - **computeTimeStep()** *// Prepare next: 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()

# Terminology/chronology of methods in TRUST

## **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 sheme -> [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](#)

## TRUST 1.7.4 developer training session



# TRUST Baltik project Tutorial

- **Modify the cpp sources**
- **Create a new cpp class**
  - **Modify your cpp class**
  - **Adding prints**

# Exploring

## Kernel module:

### **Math (Arrays, Matrix, Vect, List)**

Framework (Problem, Domain, Equation, Time schemes,  
Fields, Operators)

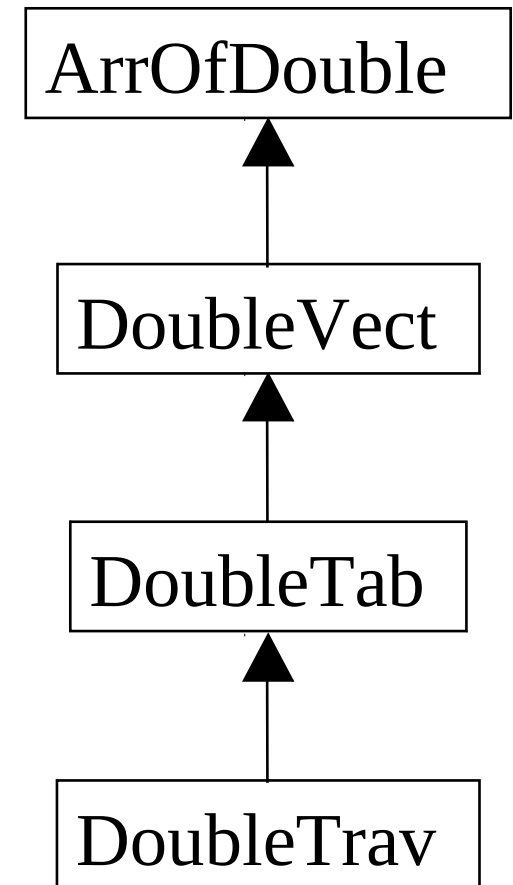
## ThHyd module

(Incompressible Thermalhydraulic)

## Space discretization module

# 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

**Notice:**

```
DoubleTab A(B); // A has the same dimensions as B, B is copied in A  
DoubleTrav A(B); // A has the same dimensions as B, A is initialized to 0 !!!
```

**Use** TRUST 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 B(A) ; // Dimension B and B=A
```

```
B+=A ; // B(i)=A(i)+A(i)
```

// not recommended:

```
DoubleTab C(n) ;
```

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

# 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
```

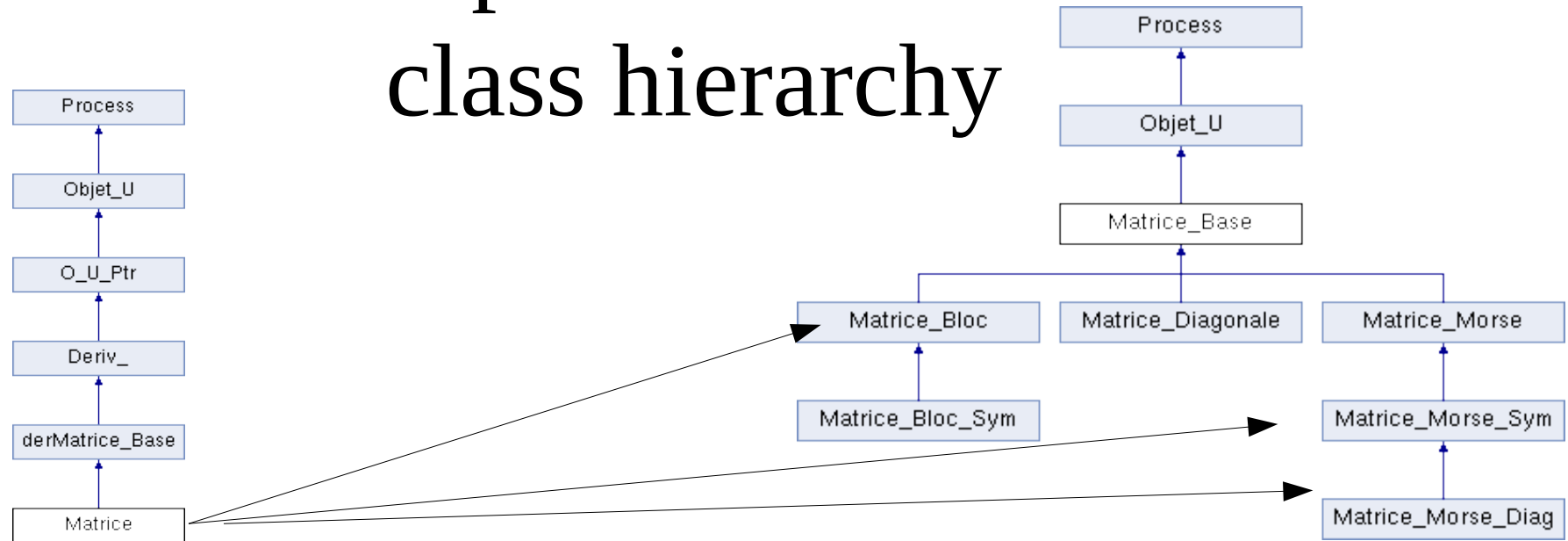
```
Cerr << A.size_array() << finl ; // n*m
```

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

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



# 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:

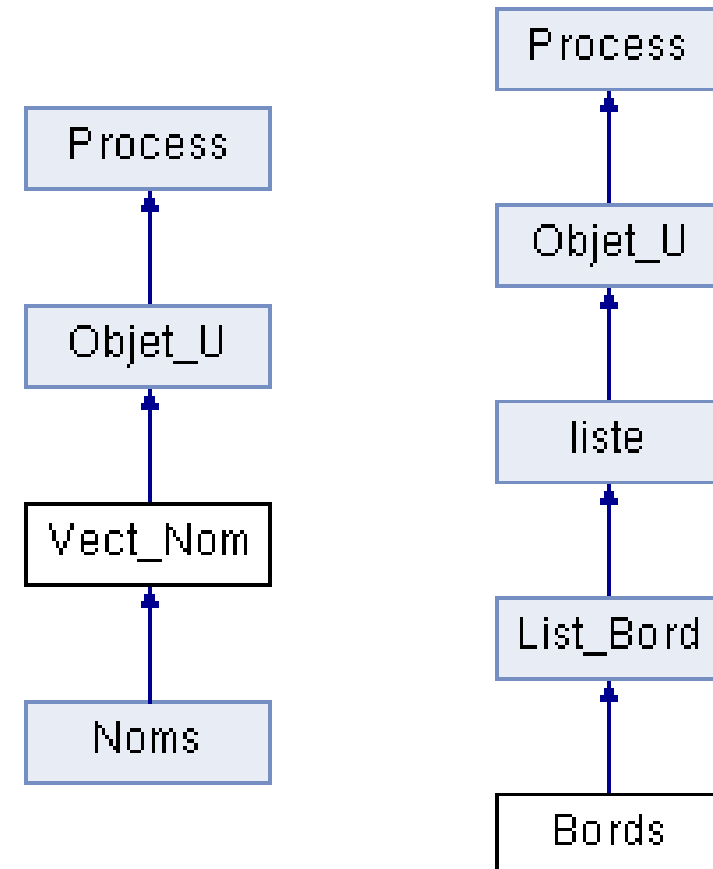
- either VECT, vector of objects
- or LIST, list of objects

Similar interface (search(), add(),...) and performances.

# 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)  
{ ... }
```

## Exercise

### Use HTML doc

Use the **HTML documentation** or **Eclipse** to see **Noms class** (have a look at the VECT methods and MacVec.h file).

→ 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) ;  
Cerr << « The number of students is » << StudentNames.??? << finl ;
```

# Exploring

## Kernel module:

Math (Arrays, Matrix, Vect, List)

**Framework (Problem, Domain, Equation, Time schemes,  
Fields, Operators)**

## ThHyd module

(Incompressible Thermalhydraulic)

## Space discretization module

# 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

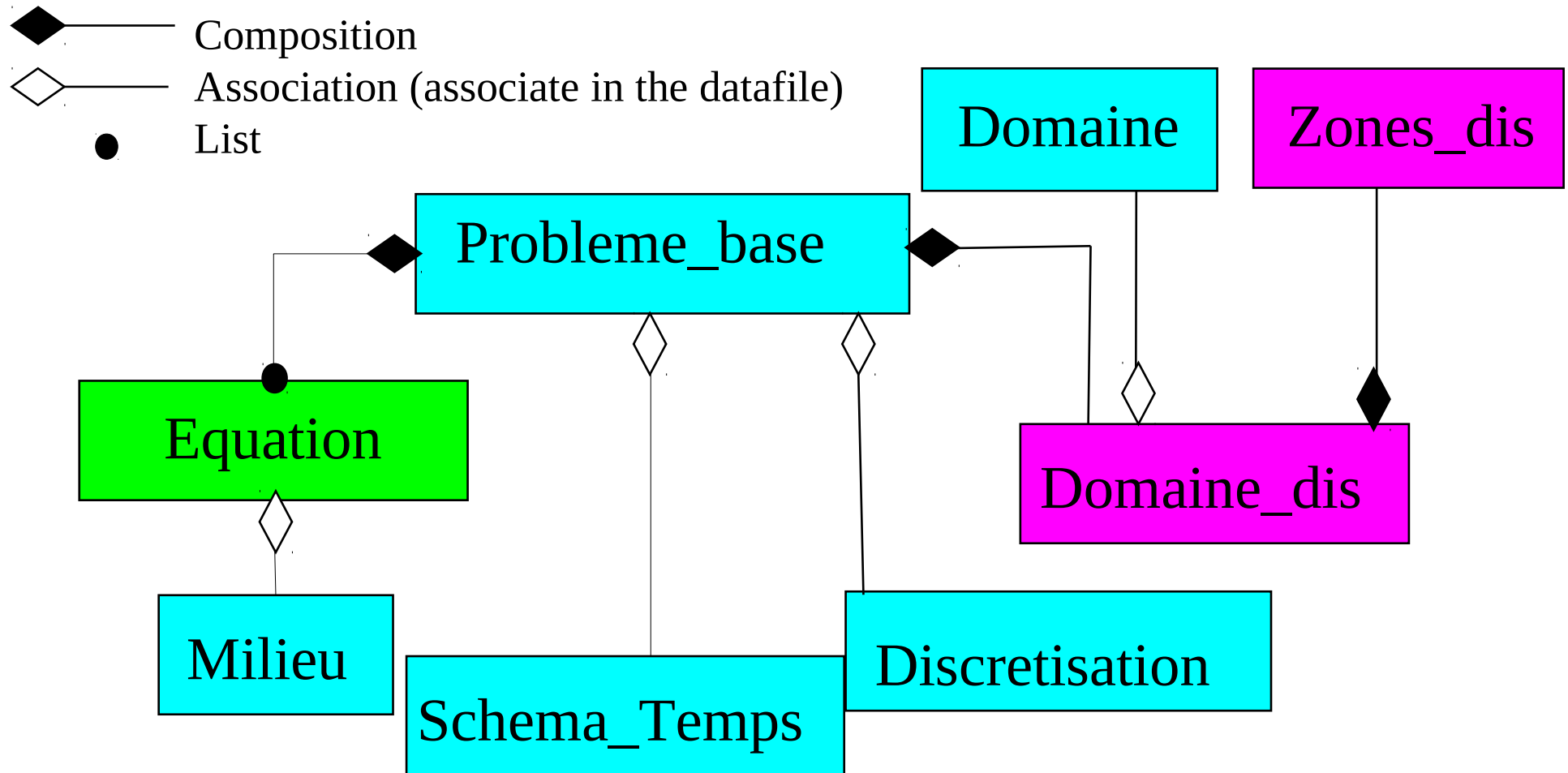
Milieu

Schema\_Temps

Discretisation

Probleme\_base

# Problem (Kernel framework)

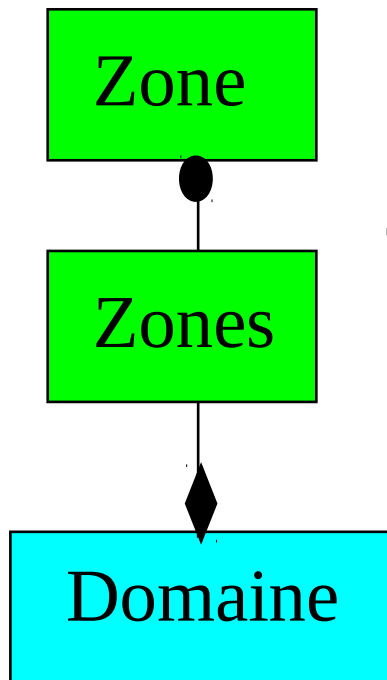




# Objects creation

- 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**

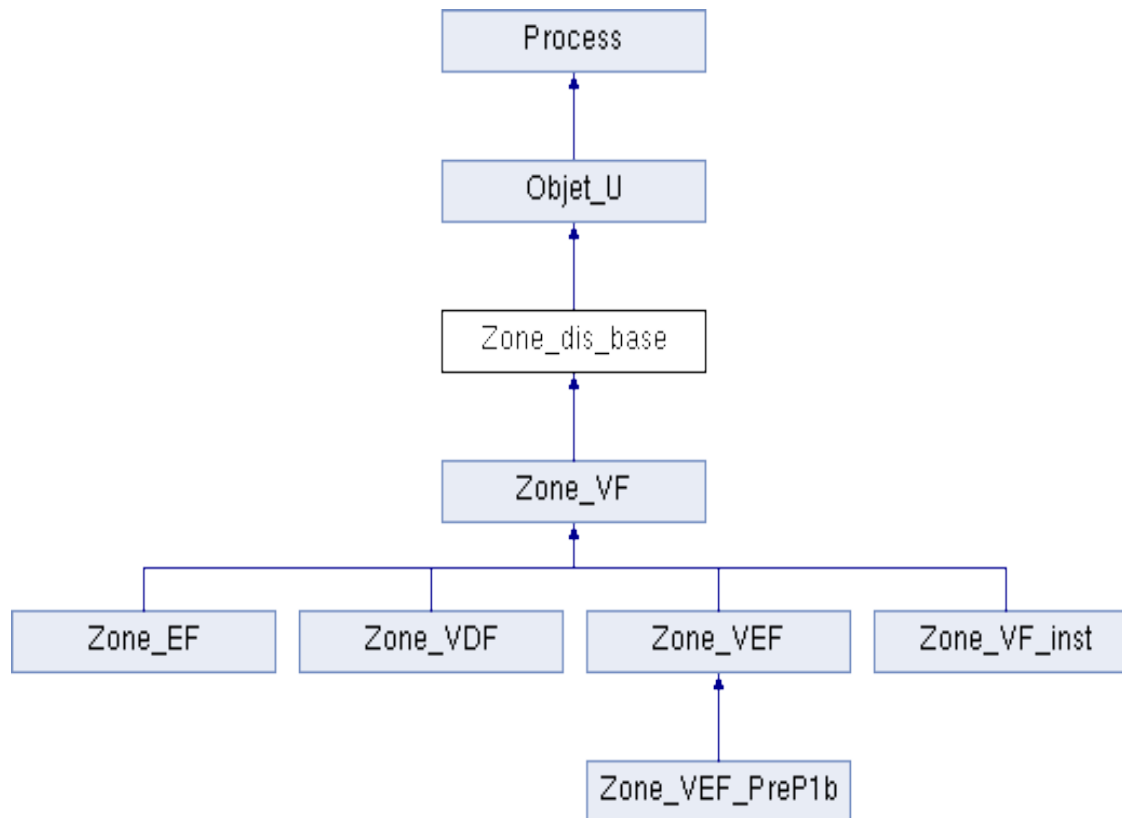
**Zones**: List of meshes to support multi meshes domain (not fully implemented in TRUST, so everywhere in the code a **Zones** list has a size of 1).

**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\_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)

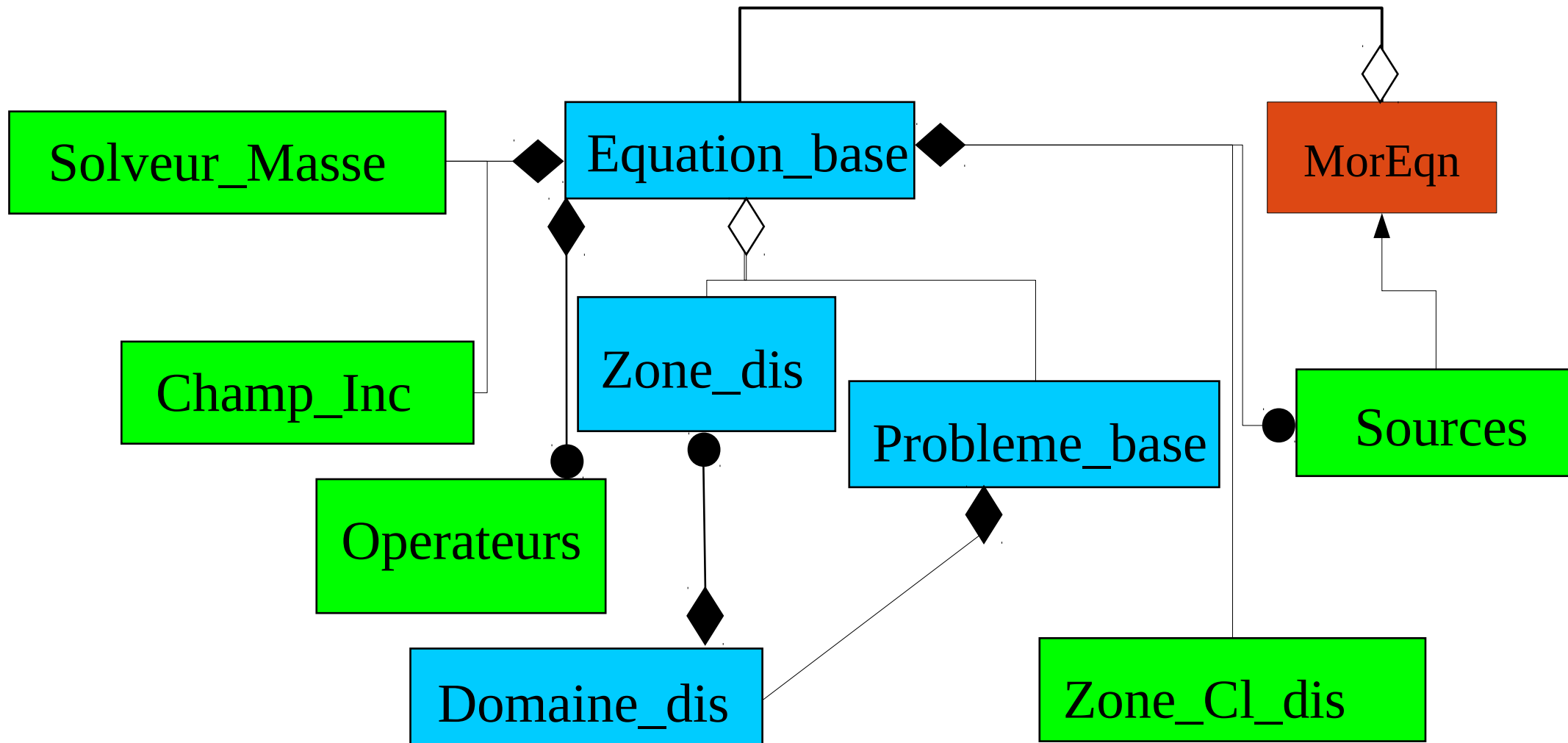
# TRUST Baltik project Tutorial

## → **Modify the cpp sources**

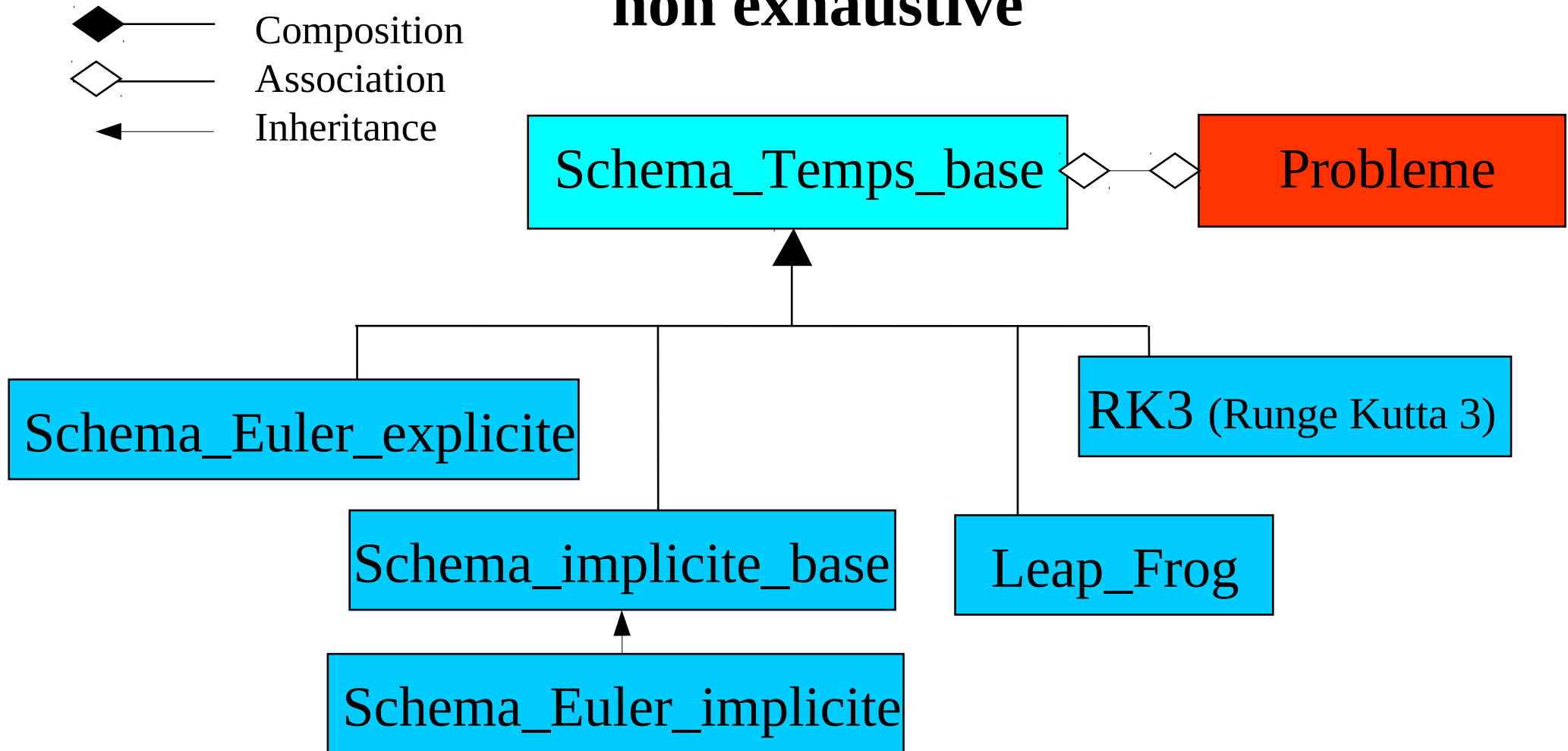
- Create a new cpp class
- **Modify your cpp class (Part 1)**
- Adding prints

# Equation (Kernel framework)

**non exhaustive**

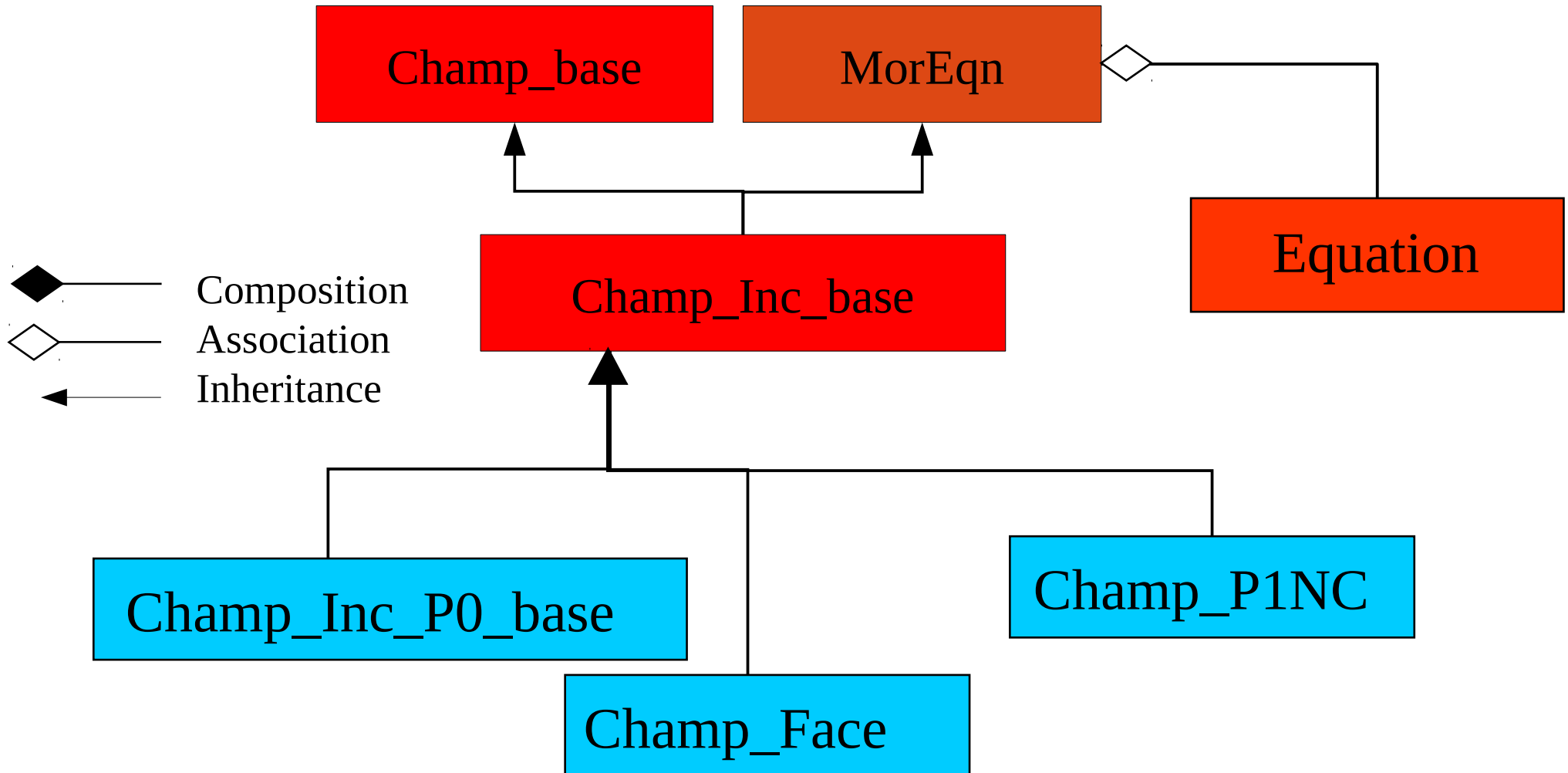


# Time Schemes non exhaustive



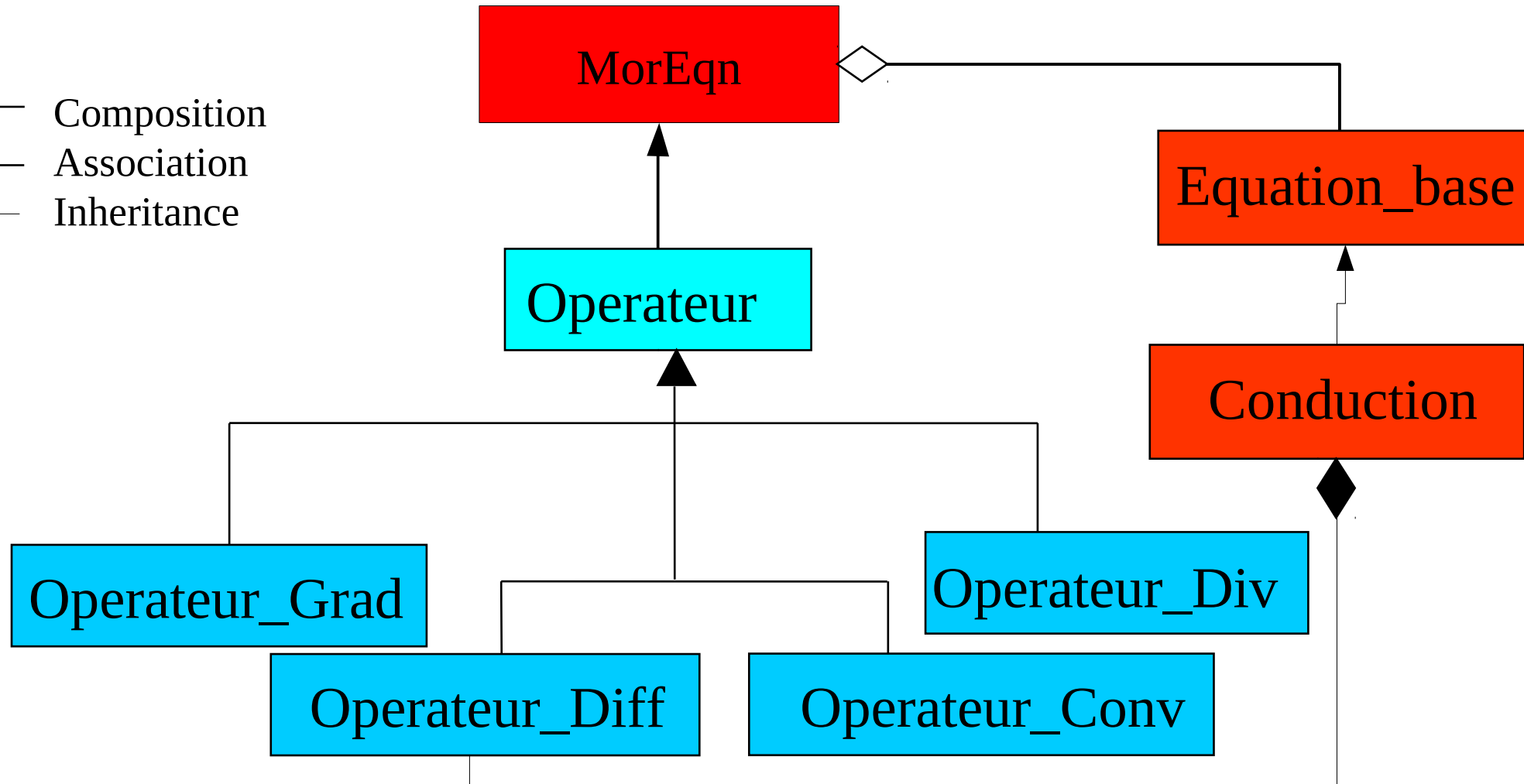
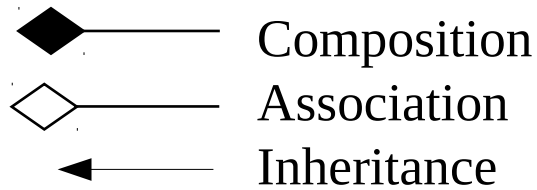
# Fields

## non exhaustive





# Operators non exhaustive



# Exploring

## Kernel module:

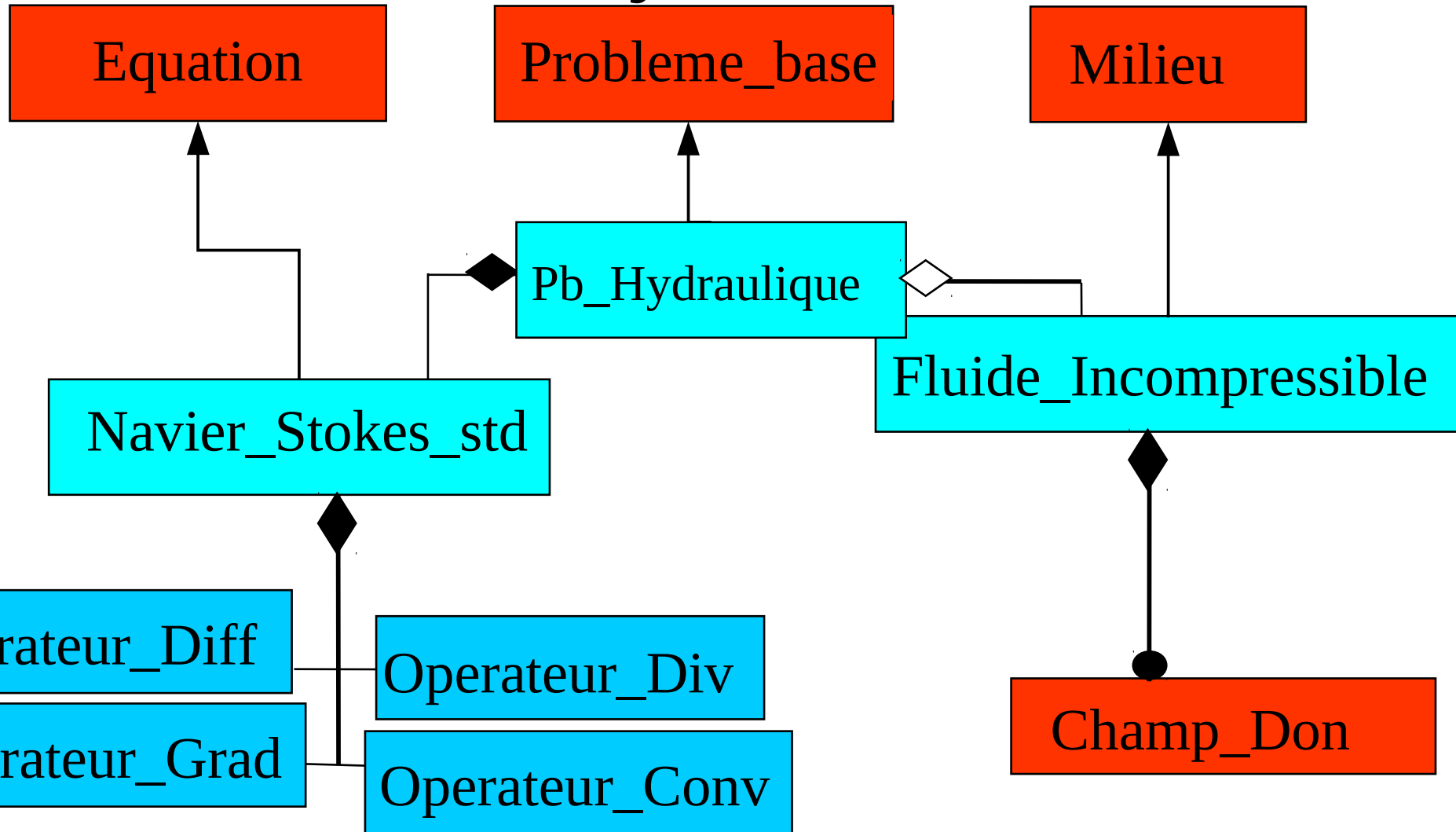
Math (Arrays, Matrix, Vect, List)  
Framework (Problem, Domain, Equation, Time schemes,  
Fields, Operators)

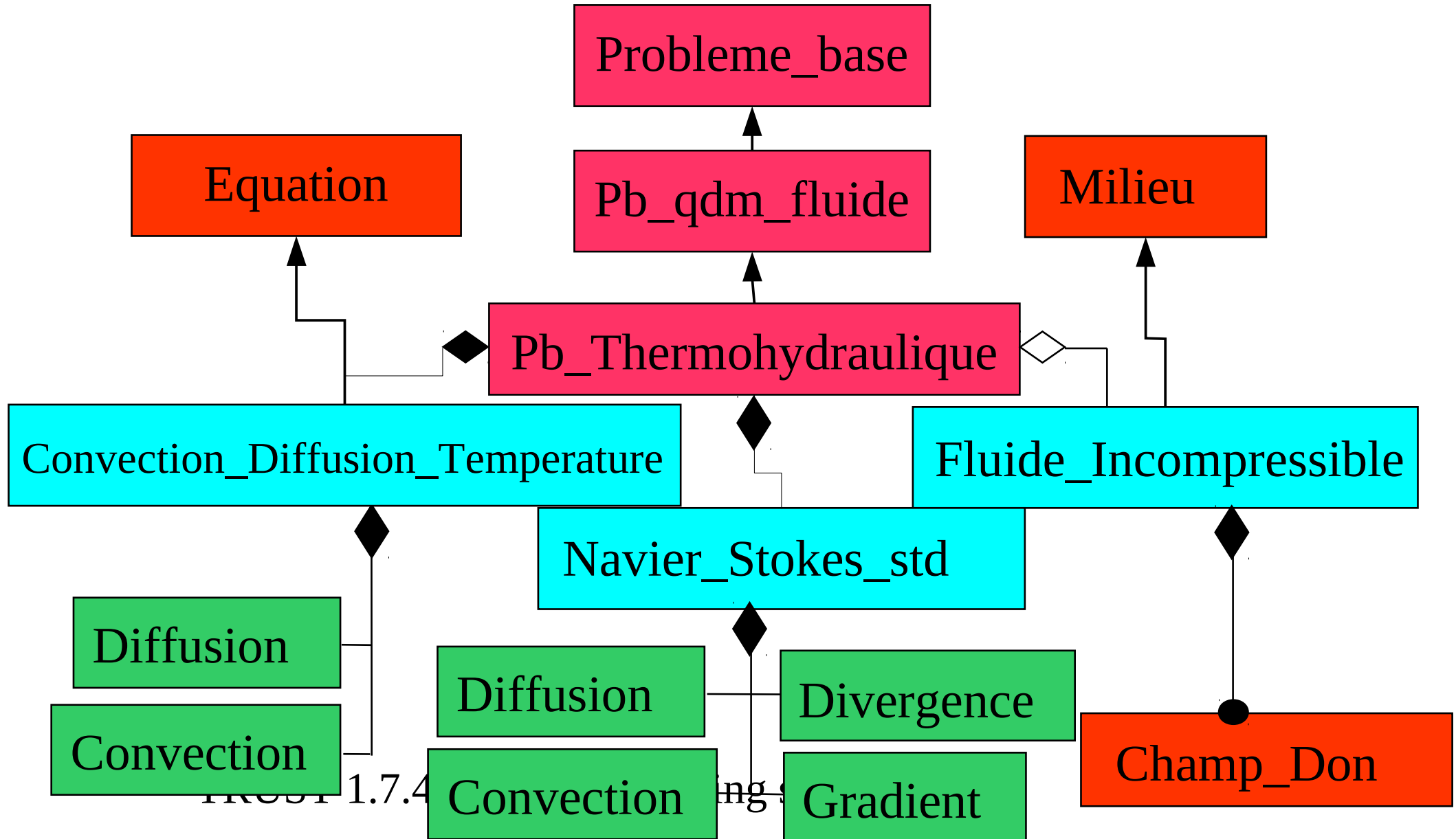
## ThHyd module

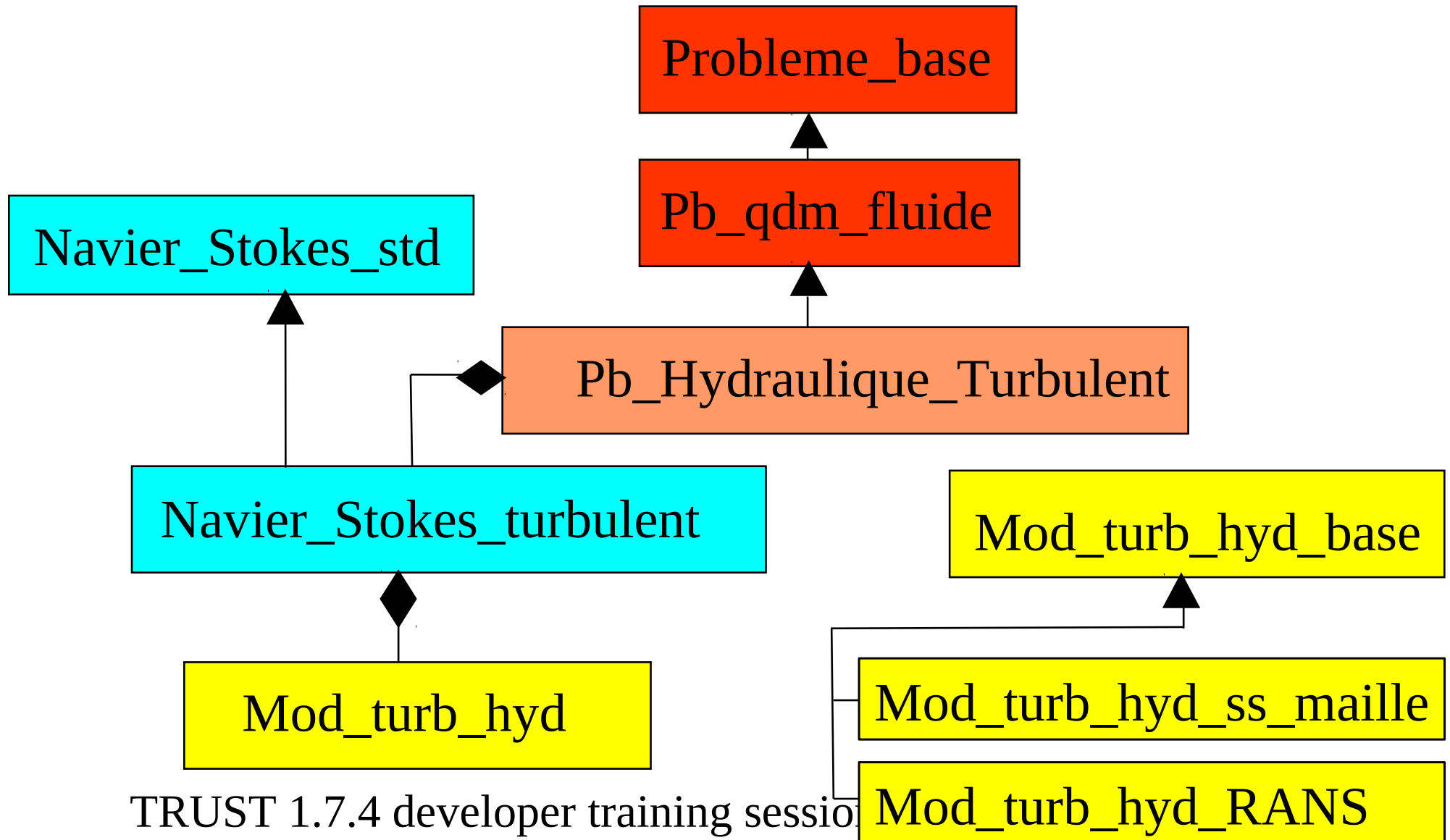
(Incompressible Thermalhydraulic)

## Space discretization module

# Hydraulic







# Exploring

## Kernel module:

Math (Arrays, Matrix, Vect, List)  
Framework (Problem, Domain, Equation, Time schemes,  
Fields, Operators)

## ThHyd module

(Incompressible Thermalhydraulic)

## Space discretization module



# Reference's Documentation

VDF: Finite-volume differences method

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

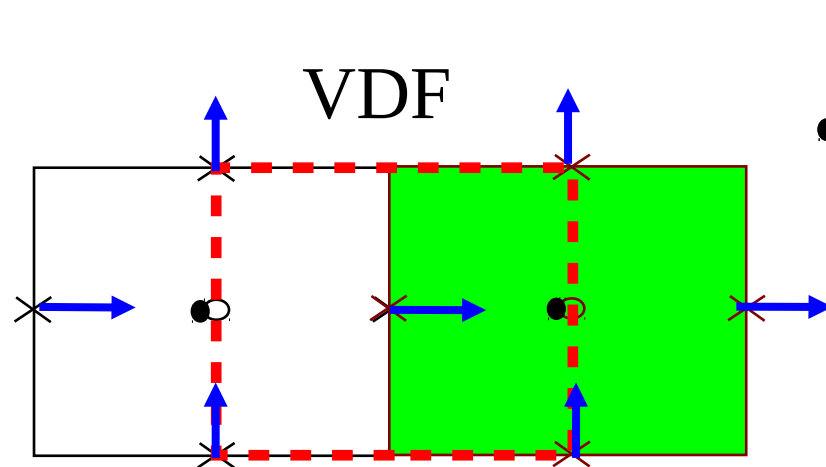
VEF: Finite-volume elements method

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



**TrioCFD website** with other PhD Thesis and articles:

<http://www-trio-u.cea.fr> → **More information on numerical methods**

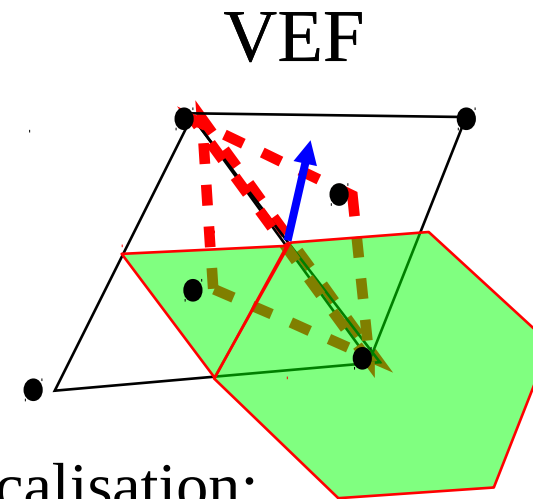
# Available discretizations




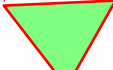
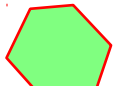
## Field localisation:

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

↑ Velocity  
• Pressure

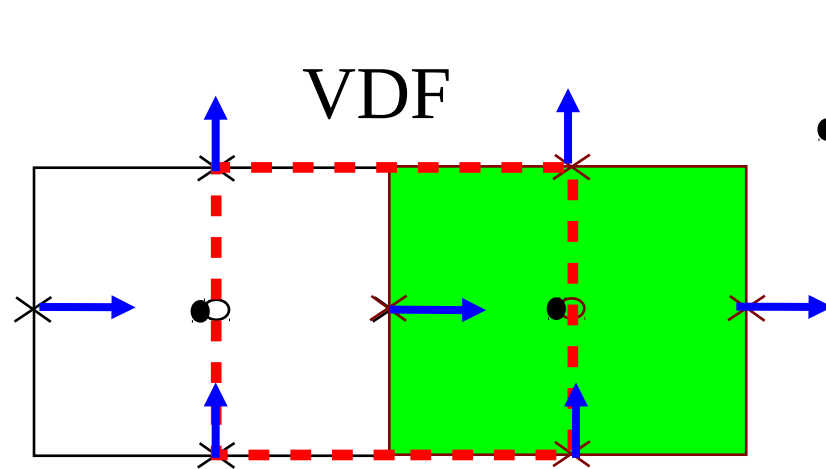


## Field localisation:

- 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



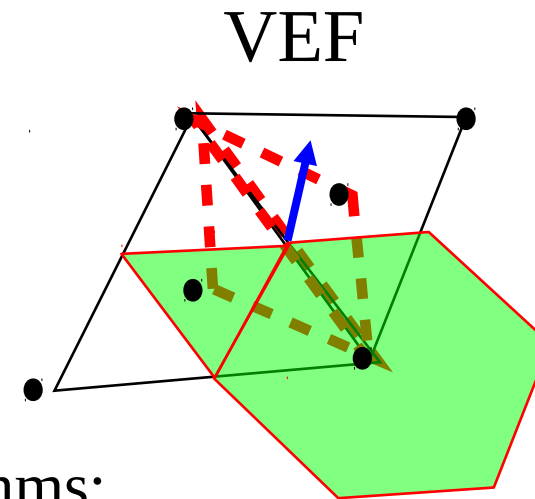
## Algorithms:

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

VDF/Operateurs/Iterateurs



VDF/Operateurs/Evaluateurs

↑ Velocity  
• Pressure



## Algorithms:

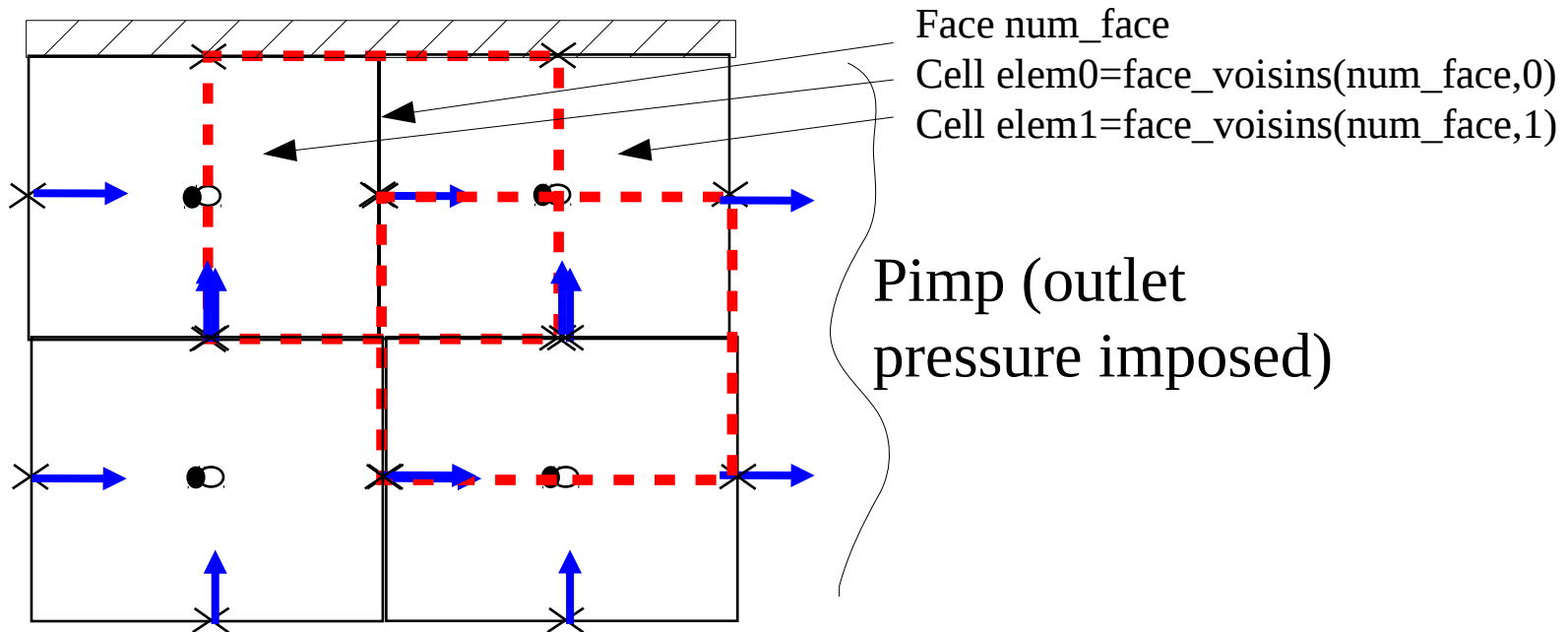
- Repeated loops on elements, faces or facets to calculate fluxes on the control volumes for each scheme

 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

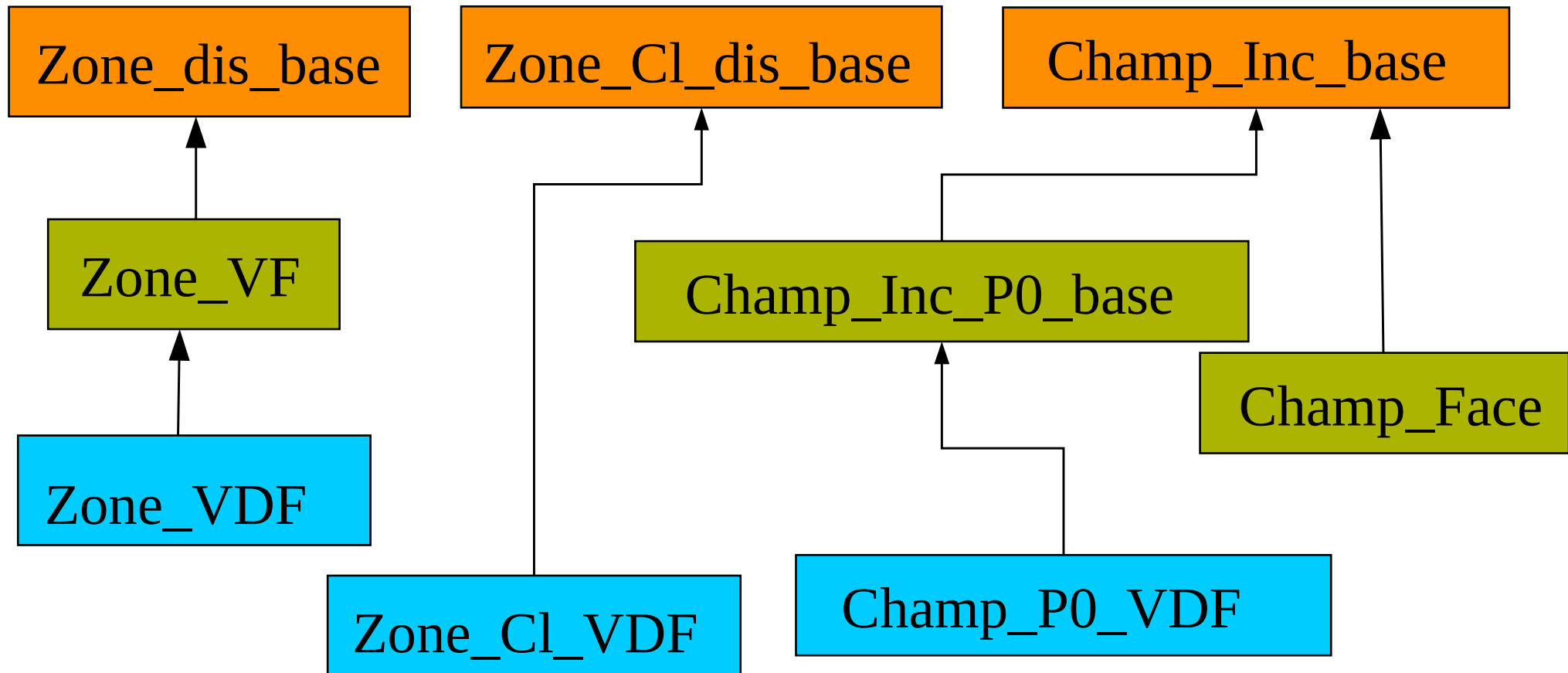
2) Loop on the internal faces :

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

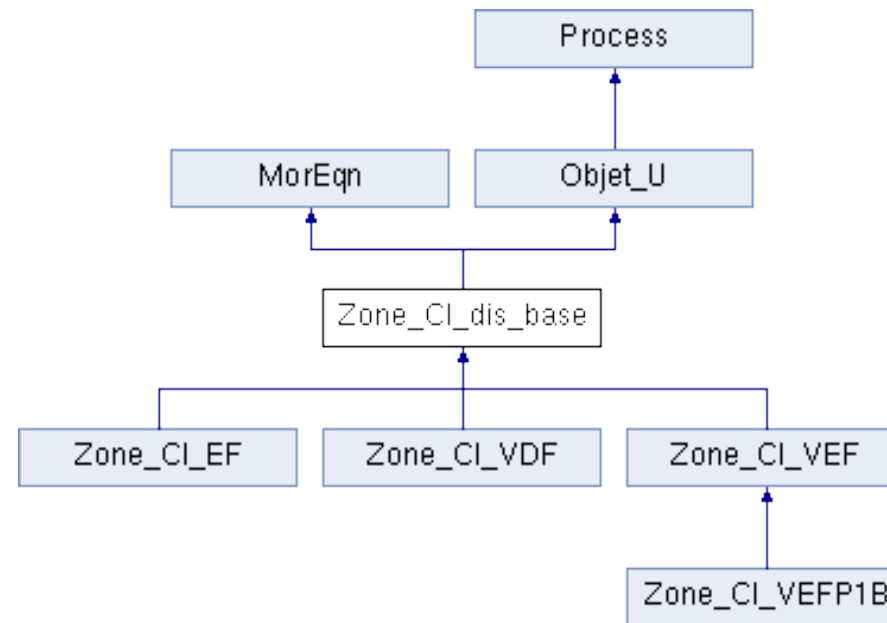
**nb\_faces()** returns the number of faces of the zone

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

# VDF Zones and Fields



# Zone\_Cl\_dis\_base

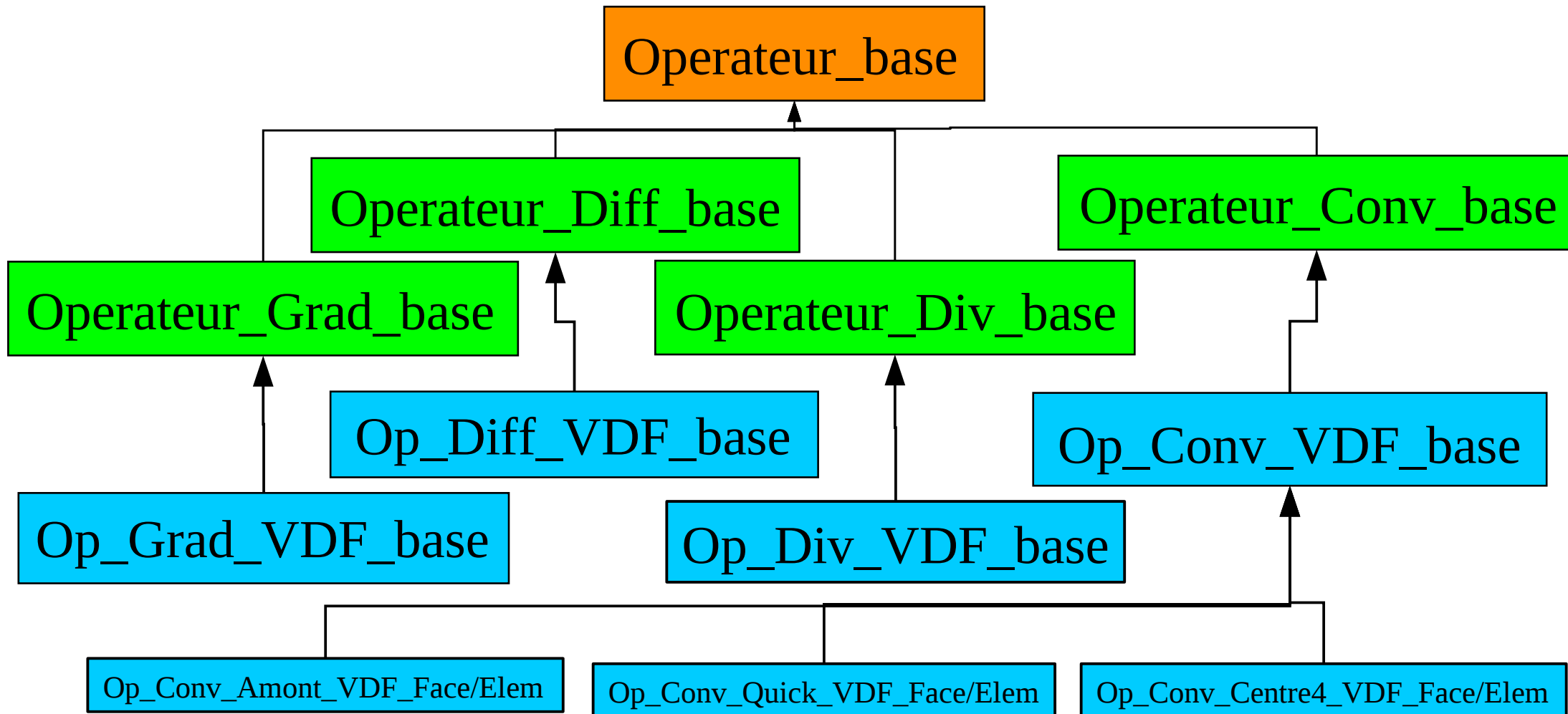


The **Zone\_Cl\_dis\_base** classe describes discretized boundary conditions :

Protected :

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

# Operators VDF implementation



# TRUST Baltik project Tutorial

## → Modify the cpp sources

- Create a new cpp class
- **Modify your cpp class (Part 2)**
- Adding prints

# How to parallelize in TRUST

Managing input/output files

Parallelism



# Parallelism

- SPMD (Single Program, Multiple Data)
- Definitions of the TRUST 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)

# How to parallelize in TRUST

## Managing input/output files

### Parallelism

# 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 ;
```

```
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;
```

# 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 »); // TRUST class to read a file  
is >> a ; // Read the 2 attributes from a file  
           // '>>': call the readOn method  
Cerr << a << finl ; // Print the 2 attributes of a  
                  // '<<': call the printOn method  
SFichier os(« newfile.txt ») ;  
os << a ; // Write the 2 attributes of a in a new file
```

# readOn - printOn

- \* “Cout”  $\Leftrightarrow$  `std::cout` on the master process only

Use this output for infos about the physics (convergence, fluxes,...)

- \* “Cerr”  $\Leftrightarrow$  `std::cerr` on the master process only

Use this output for warning/errors only

- \* “finl”  $\Leftrightarrow$  `std::endl + flush()` on the master process

- \* “Journal()” prints to "datale\_000n.log" files.

Use this output during parallel development to print plumbing infos which would be hidden during later production runs.

# TRUST Baltik project Tutorial

## → Modify the cpp sources

- Create a new cpp class
- Modify your cpp class
- **Adding prints**

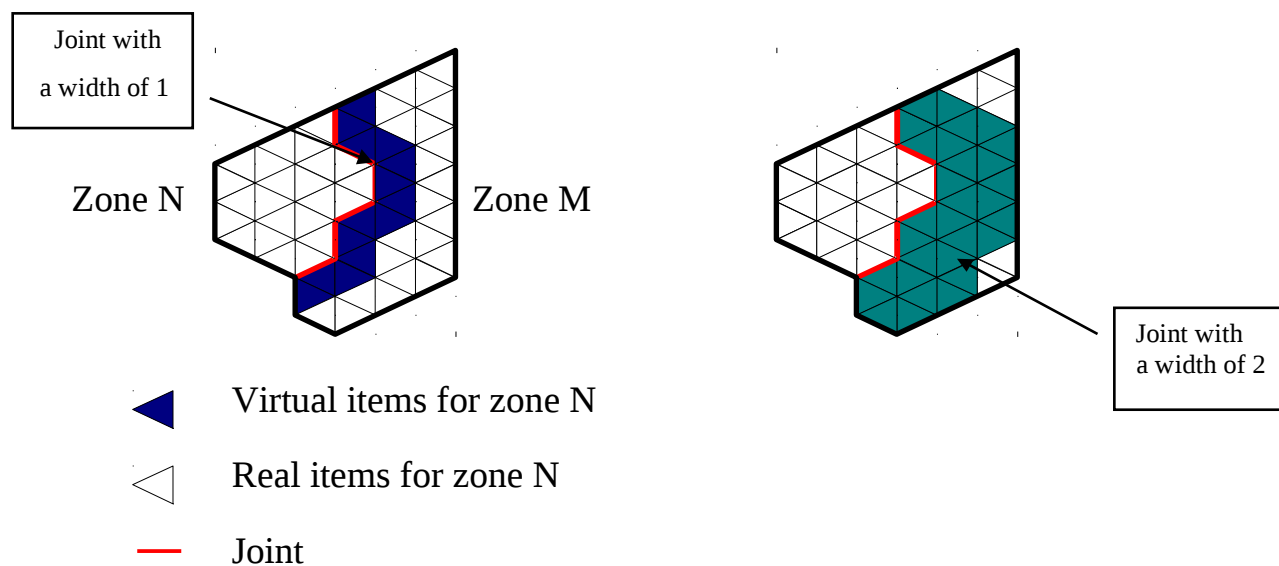
# How to parallelize in TRUST

Managing input/output files

**Parallelism**

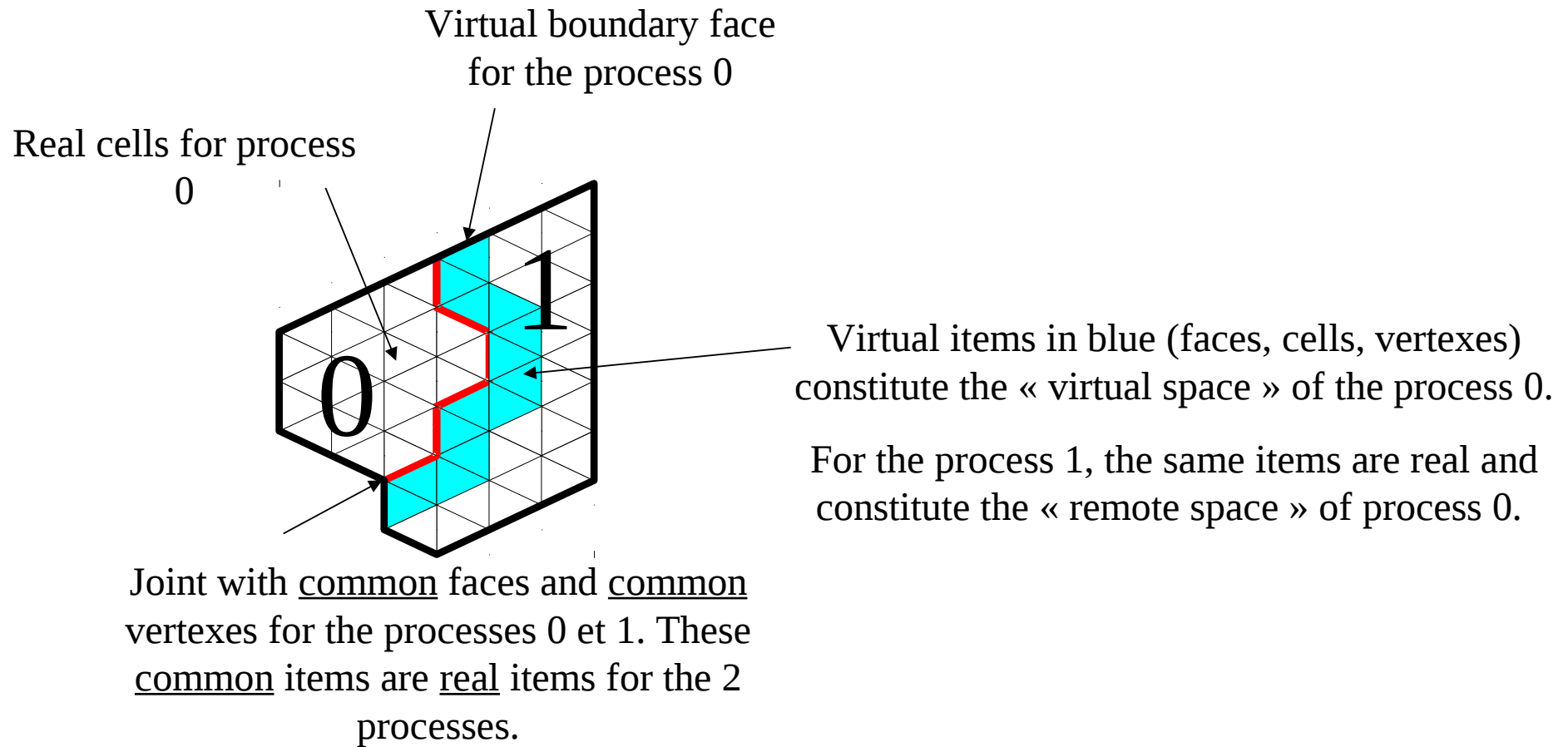


# Parallelism



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()

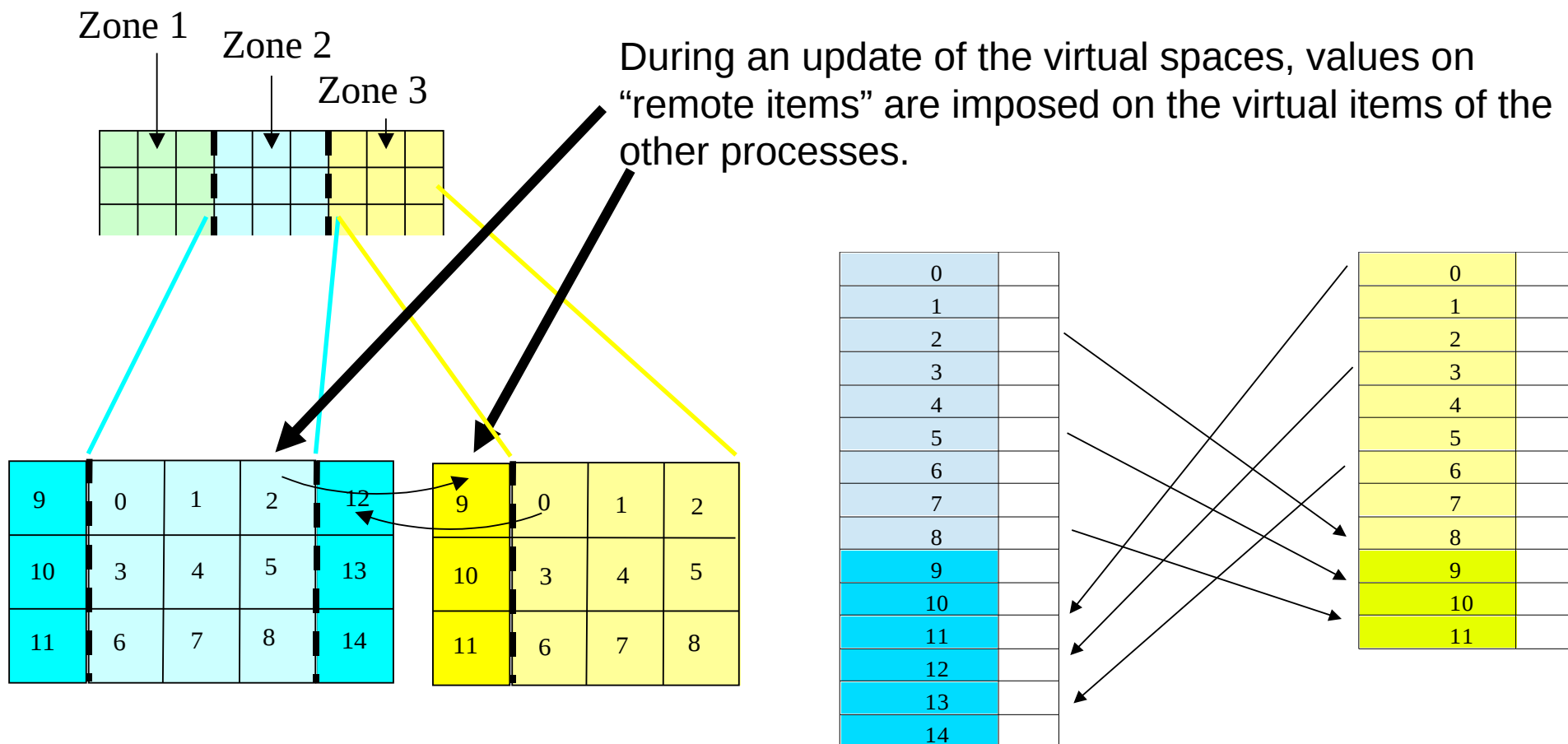
Zone::nb\_elem\_tot()

1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	

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

# Parallelism

## Example of a distributed array on cells



# Parallelism

- 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));`

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

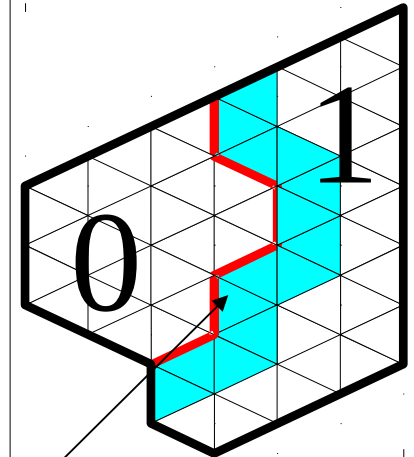
- It depends of your algorithm and the items you are using
- Use carefully distributed arrays. It will slow down the parallel execution during each virtual spaces update
- Example where you need it: You want to calculate the interpolation of a cell centered field to the faces of the mesh :





# 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 TRUST 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**(DoubleVect& x) 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
  - Journal() : all the processes write to the .log files

# Parallelism

- Send/receive methods (envoyer/recevoir). Well described in the file :
  - \$TRUST\_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

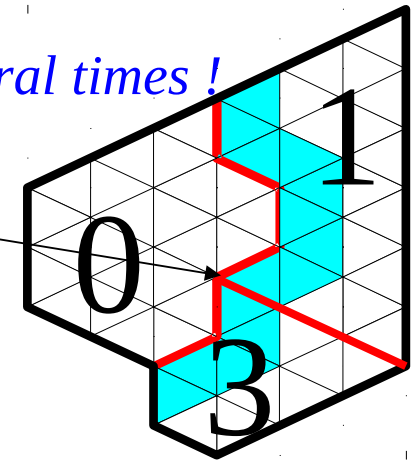
– Pitfall with the common items :

*/\* 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)** ;

# Parallelism

-Pitfall with how the faces are ranked in TRUST (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())

0	
1	
2	
3	
4	
5	
6	
...	
nb_faces_int()-1	
nb_faces_int()	
nb_faces()-1	
nb_faces()	
nb_faces_tot()-1	
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

# TRUST Baltik project Tutorial

## → Parallel exercise :

- Part 1
- Part 2 Optional
- Part 3
- Part 4 Debog



# Parallelism

## – How to validate parallelization in TRUST

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):

**trust datafile**

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

**trust parallel\_datafile  $N$**

**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 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

# TRUST Baltik project Tutorial

## → Parallel exercise :

- Part 1
- Part 2 Optional
- **Part 3**
- Part 4 Debog

# Parallelism

## – How to debug parallelization in TRUST

- 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 :  
**exec=\$exec\_debug trust datafile N**
- What if the parallel calculation crashes/hangs ?
  - Give a try with the debugger to know exactly where the issue is :  
**exec=\$exec\_debug trust -gdb datafile N**

# Parallelism

## How to find the source(s) of parallelism differences in TRUST?

-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.

# TRUST Baltik project Tutorial

## → Parallel exercise :

- Part 1
- Part 2 Optional
- Part 3
- **Part 4 Debog**

# TRUST test coverage

# Code coverage

- Created by gcov tool, as a nightly task on ~2000 test cases.
- 70% of TRUST & his Baltiks 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.
- TRUST/TrioCFD code coverage and tools exploiting it are available for the developer



# Useful code coverage tools

- TRUST tool to know and run the test cases covering a method:

**trust -check class::method**

*Example :*

```
$ trust -check Navier_Stokes_std::mettre_a_jour
```

```
$ nedit liste_cas
```

- To check the non-regression on one or several test cases

**trust -check all|testcase|list**

Some examples for Baltik developer:

```
$ make check_optim|check_debug # Check the project non-regression on Baltik test cases
```

```
$ make check_last_pb_debug # Running last pb test suite (see test in liste_pb.all file)
```

```
$ make check_deps_debug # Check the project non-regression on dependencies test cases
```

```
$ make check_all_debug # Check the project non-regression on all test of project = baltik test + dependencies test
```

```
$ make check_trust_optim # Check the project non-regression on TRUST test cases
```

```
$ make check_full_debug # Check the project non-regression on full test suite (all test of project + all test of TRUST platform)
```

# How to debug TRUST

gdb  
valgrind



# Use gdb tool to debug or understand the code

## GDB web site and documentation:

<https://www.gnu.org/software/gdb/>

<https://doc.ubuntu-fr.org/gdb>

## Online tutorials:

<http://www.linux-france.org/article/memo/node119.html>

[http://perso.ens-lyon.fr/daniel.hirschhoff/C\\_Caml/docs/doc\\_gdb.pdf](http://perso.ens-lyon.fr/daniel.hirschhoff/C_Caml/docs/doc_gdb.pdf)

With TRUST, run with Eclipse or in a terminal:

# To describe all the commands:

**\$ man gdb**

# To debug the TRUST binary program compiled with -g:

**\$ exec=\$exec\_debug trust -gdb datafile**

# Use gdb tool to debug or understand the code

## # 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 TRUST 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 TRUST (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:**

\$ make\_PAR.data datafile N

\$ exec=\$exec\_debug trust -gdb PAR\_datafile N

# Use valgrind to find memory bugs

- Valgrind is a memory checker tool: <http://www.valgrind.org>
- You can check a binary with:  
**\$ VALGRIND=1 trust datafile**
- It detects uninitialized variables, memory leaks, outbound array values,...
- TRUST 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,...)

# TRUST Baltik project Tutorial

- **Code coverage exercise**
- **Tools**
  - **GDB exercise**
  - **Use Valgrind to find memory bugs**

# TRUST 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 all modules
- All news (classes, keywords, ...) in english

# Rules to contribute

You want your work to be merged in the next release of the TRUST,  
then provide to the TRUST support team :

## If you develop in a Baltik project based on TRUST:

- 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\_full\_debug # Check the project non-regression on full test suite (all test of project + dependances + TRUST)
  - VALGRIND=1 make check\_optim # Same in optimized mode with Valgrind check

# After the training session...

Read the commented solution of the exercise:

[\\$TRUST\\_ROOT/doc/TRUST/exercices/my\\_first\\_class](#)

Practice on a tutorial:

[\\$TRUST\\_ROOT/doc/TRUST/exercices/equation\\_convection\\_diffusion/rapport.pdf](#)

Or

**trust -index** → « Other baltik tutorial »

The End

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