Université Libre de Bruxelles

von Karman Institute for Fluid Dynamics
Aeronautics and Aerospace Department

PhD. Thesis

An Object Oriented and High Performance Platform for Aerothermodynamics Simulation

Andrea Lani

November 25, 2008

Promoter: Prof. Herman Deconinck

Contact information:
Andrea Lani
von Karman Institute for Fluid Dynamics
72 Chaussée de Waterloo
1640 Rhode-St-Genèse
BELGIUM

email: lani@vki.ac.be

This manuscript was typeset on LaTeX with KOMA-Script.

All the research work was done with the GNU/Linux operating system.

Grazie alla mia famiglia per il
continuo affetto e supporto morale

Un ringraziamento speciale per Isabel,
che, con la sua infinita pazienza, mi è stata vicina
in tutti i momenti, specialmente nei più difficili

Summary

This thesis presents the author's contribution to the design and implementation of COOLFluiD, an object oriented software platform for the high performance simulation of multi-physics phenomena on unstructured grids. In this context, the final goal has been to provide a reliable tool for handling high speed aerothermodynamic applications.
To this end, we introduce a number of design techniques that have been developed in order to provide the framework with flexibility and reusability, allowing developers to easily integrate new functionalities such as arbitrary mesh-based data structures, numerical algorithms (space discretizations, time stepping schemes, linear system solvers, ...), and physical models.
Furthermore, we describe the parallel algorithms that we have implemented in order to efficiently read/write generic computational meshes involving millions of degrees of freedom and partition them in a scalable way: benchmarks on High Performance Computing (HPC) clusters with up to 512 processors show their effective suitability for large scale computing.
Several systems of partial differential equations, characterizing flows in conditions of thermal and chemical equilibrium (with fixed and variable elemental fractions) and, particularly, nonequilibrium (multi-temperature models) have been integrated in the framework.
In order to simulate such flows, we have developed two state-of-the-art flow solvers:

1. a parallel implicit 2D/3D steady and unsteady cell-centered Finite Volume (FV) solver for arbitrary systems of Partial Differential Equation (PDE) on hybrid unstructured meshes;

2. a parallel implicit 2D/3D steady vertex-centered Residual Distribution (RD) solver for arbitrary systems of PDE's on meshes with simplex elements (triangles and tetrahedra).

The Finite Volume (FV) code has been extended to handle all the available physical models, in regimes ranging from incompressible to hypersonic. As

far as the Residual Distribution (RD) code is concerned, the strictly conservative variant of the RD method, denominated Conservative Residual Distribution (CRD), has been applied for the first time in literature to solve high speed viscous flows in thermochemical nonequilibrium, yielding some preliminary outstanding results on a challenging double cone flow simulation.

All the developments have been validated on real-life testcases of current interest in the aerospace community. A quantitative comparison with experimental measurements and/or literature has been performed whenever possible.

Contents

Acknowledgements

At the end of those many years of doctorate, after a total of 6 and a half years at the Von Karman Institute, I can sincerely say that the list of people to whom I owe my gratitude is really huge. I'll do an effort not to forget anybody and I apologize in advance if this unfortunately happens ...
First of all, I would like to thank my promoter and supervisor Prof. Herman Deconinck, that since we first met, at the end of my short experience in NUMECA, has always expressed his trust in me. His being supportive and enthusiastic about every little progress I made or result I got has really motivated me to constantly improve and set the "bar" a little bit higher.
Secondly, I'll be eternally grateful to Koen and David, my former colleagues in NUMECA and two of the brightest persons I have ever known, who literally push me into the von Karman Institute for Fluid Dynamics (VKI) experience. It was really a pleasure to see David back as a colleague at the institute. He has been a huge source of inspiration for me (and many others) and his "can-do" attitude has served as stimulation for boosting my work.
Next in the list, at least chronologically, comes my COOL-mate Tiago: I will never forget that October 2003, when we sat everyday upstairs, in the computer room, next to each other like siamese twins with our own laptops and, despite our occasionally divergent opinions, we gave birth to the first working embryo of COOLFluiD. Well, the rest is recent history ... in three words : "We made it!". I really hope to keep on collaborating with you to make grow "our baby" even more.
Thanks to Dries who introduced me into the realm of parallel programming: our design COOL-meetings in VKI or KU Leuven or our short trips together (while coming to VKI in my car) have been a great occasion to exchange fruitful ideas. I will never thank you enough for all the assistance you gave me in running large simulations on KU Leuven cluster, even during holidays! Many thanks to Kurt, Jirka and Mario who have answered many of my tedious Computational Fluid Dynamics (CFD) questions at the beginning of my project. A special thanks goes to Telis: I was literally addicted to your meshes and you had a great patience to help me also with my Linux troubles whenever I asked you. I cannot speak of Linux without mentioning

Giuseppe, Federico and Raimondo, the computer center shamans, who gave me their support in a countless number of occasions.
Many thanks to Marco whose close collaboration helped me to move a huge step further towards the final goal and often succeeded in distressing me with his constant good mood. Your contribution has been fundamental for achieving the main results presented in this thesis and I really would like to continue working together also in the future, despite the probable long distance.
Thanks to Carlo for the relaxed lunch breaks and long chats in some of the few restaurants in the VKI neighborhood and his technical assistance for my occasional car troubles!
Thanks to Damiano, Alberto and Khalil for our regenerating tennis matches; to Thierry for his advices and the great enthusiasm that he can always transmit; to Thomas (Nierhaus) for some regenerating chats on heavy metal CDs or concerts; to Karin, Nadege and Thomas for having being patient office mates. In particular, additional thanks to Thomas for having given such a shot to the COOLFluiD project, with his admirable capability to master the most diverse and complex applications.
Thanks to all my former students, Sarp, Michel, Janos, Radek, Fabio, Christophe, etc. who all contributed to add critical mass to this thesis.
My gratitude goes to the VKI Director, Mario Carbonaro for having accepted me in the doctoral program; to Patrick for the trust and his quick response whenever I had a need for more computational resources; to Olivier, Doug and Prof. Degrez for having expressed in so many circumstances their support for the COOLFluiD project.
Last but not least, I would like to say "Grazie mille" to my family who have always loved me, believed in me and encouraged me, especially during the hardest times. Thanks to Isabel for her love, infinite patience and support: you, more than anybody else, know who is the guy and the effort behind this thesis!

Chapter 1

Introduction

In the scientific field, it often happens that a researcher or a team of researchers start writing a software package that is meant to tackle some specific problems. When new problems or new methodologies to solve problems pop up, one has to choose between adding new features into an existing code or writing a new one from scratch.
The first solution can easily lead to an unforeseen and hardly manageable growth of the original software and can result in a series of frequent changes, re-factorings and, in the worst case, full re-design.
The last solution is typically adopted when the needed changes are incompatible with the existing software or when the desired new features are far from the original goals that were planned in the long-term.
In both cases, the importance of allowing and maximizing the software reuse and modularity becomes clear. The question that arises is then: how to maximize the reusability and the flexibility of the software? To answer these needs, a great help has been given to the scientific community during the last 15 years by the Object-Oriented (OO) programming. The latter has pushed developers to improve their way of conceiving and designing software by redirecting the focus more on the way scientific concepts and algorithms interact than on the mere functionalities per se.
At the same time, the increasing power of computer architectures and the concurrent advance towards parallel and distributed computing have allowed researchers to perform more and more challenging simulations of complex physical phenomena, motivating at the same time the development of increasingly sophisticated numerical algorithms.
As an example, CFD represents a field in which powerful computer resources have become fundamental to be able to simulate complex flow fields closer to the demands of industry, medicine and, more generally, technological progress. The required power is not anymore only a matter of execution speed or memory capacity, but it has also become a matter of software design, since scientific researchers need constantly the implementation of new features, models, tools to increase the quantity and enhance the quality of

their results.
Today's CFD applications require a considerable degree of flexibility from numerical tools, since all of the following situations can occur and should be envisioned:

- the same physical phenomena can be solved by means of different space discretization methods (Finite Volume, Finite Difference, Finite Element, Residual Distribution, Discontinuous Galerkin ...), time integrators, boundary condition treatments, iterative and direct linear system solvers etc.;
- the same numerical methods can be applied to different physical problems and their corresponding set of equations;
- some subsets of the governing equations may need to be treated with different numerical schemes;
- different physical models and constitutive relations can be applied to close the same set of governing equations.

Moreover, data structures of research codes have to be flexible and dynamic enough to support parallelization, mesh adaptation, multigrid techniques.
In order to get all these capabilities in one single code, the commonly used procedural programming based on functional decomposition has proven to be insufficient. On the contrary, OO design and programming have been of great help in tackling all these rapidly increasing needs in a more appropriate and scalable way. During the last decade, this has lead to the development of several platforms for research purposes such as DiffPack [6], Deal II [12], LibMesh [70], Life [130], OpenFoam [44]), etc. and the migration of most commercial flow solvers (Fluent, StarCD, Numeca's FINE ...) to the new approach.

1.1 The COOLFluiD project

1.1.1 History of the project

The current work has moved its first steps back in March 2002, within the **COOLFluiD** (**C**omputational **O**bject **O**riented **L**ibrary for **F**luid **D**ynamics) project, aiming at creating an OO framework for the simulation of multi-physics on unstructured meshes.
One of the primary goals of COOLFluiD was to port to a new common

platform the main simulation capabilities spread over a large number of unmaintained legacy codes, developed at the Von Karman Institute by the previous generation of PhD students. Such a framework was also and especially meant to be scalable, i.e. capable of evolving, maintainable and to easily accommodate new developments for many years to come, both from the physical modeling and the algorithmic sides.
The preliminary design was the fruit of the collaboration between the author and two other PhD candidates, namely Tiago Quintino (VKI) and Pieter De Ceuninck (KU Leuven).
Between January and September 2003, Quintino and the author worked on separate tracks, each one on a different implementation of the same concept design. In October of the same year, the author's version of COOLFluiD [74] and Quintino's HOTFluiD merge into one single platform, keeping the best capabilities and properties of both codes, but bringing them to a higher level of flexibility, thanks to a partial synergic re-design and re-implementation effort.
After the definition of a unified framework and the entry in the team of Dries Kimpe (KU Leuven) and, later on, of Thomas Wuilbaut (VKI), the scope of application was progressively enlarged to high-performance computing and more and more complex multi-physics. During the last few years, also thanks to the contributions of many other PhD candidates, diploma course members, post-docs and external partners, the project has been growing very rapidly towards the status of a customizable collaborative environment for CFD, where many researchers can cooperate to extend the simulation technology towards predefined target applications while taking maximum benefit from the others' work.

1.1.2 The COOLFluiD Architecture

COOLFluiD consists of a collection of dynamically linked libraries and application codes, organized in a multiple layer structure, as shown in Fig. 1.1. Two kinds of libraries are identifiable: kernel libraries and modules. The former supply basic functionalities, data structure and abstract interfaces, without actually implementing any scientific algorithm, while the latter add effective simulation capabilities.
The framework was developed according to a *plug-in* policy, based on the self-registration [19, 74, 78] and self-configuration techniques [78, 131], which allow new components or even third party extensions to be integrated at run-time, while only the Application Programming Interface (API) has been exposed to the developers. From top to bottom of the COOLFluiD architec-

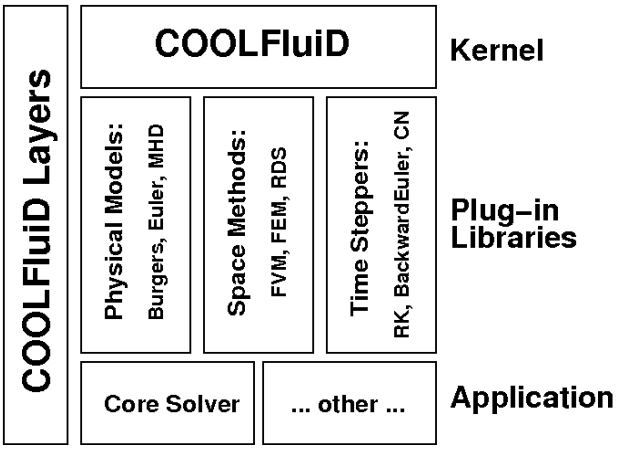

Figure 1.1: Multi-layer structure of COOLFluiD.

ture, one finds, progressively, more functional layers: the kernel defines the interfaces, the modules implement them and the application codes can select and load on demand the libraries needed for an actual simulation. Moreover, COOLFluiD is a component-based platform, where each functionality is enclosed in a separate component which can be connected to others at run time to create one or more actual application solver.

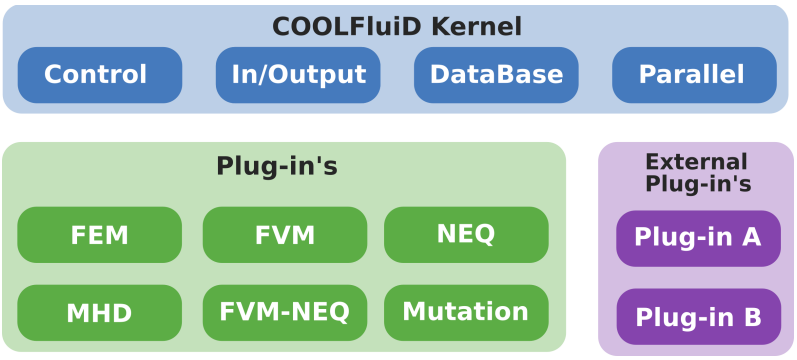

Figure 1.2: Component-based architecture of COOLFluiD: kernel modules, internal and external plug-ins.

Fig. 1.2 shows an example of some of the components available in COOLFluiD and the distinction between kernel modules, internal and external plug-ins

is stressed. Internal plug-ins are typically developed by VKI or by partners with access to the source code, while external plug-ins are usually developed independently by third parties with access only to the API. This multi-layer and multi-component structure guarantees flexibility and modularity at the highest design level and helps managing the development process, since layers and components automatically reflect different implementation levels and, in some cases, permission rights.

1.2 Objectives of the present thesis

The original goals of the COOLFluiD project have traced the guidelines for the present work, aiming at implementing a flexible and extendible object-oriented platform for high-performance simulation of aerothermodynamics on unstructured grids. This rather ambitious target has given the project a multi-disciplinary connotation nature since the start, requiring development of competence and research effort in different areas:

1. OO design tailored to scientific applications to provide the framework with a solid and flexible software infrastructure, allowing new developers to easily integrate new functionalities and fully reuse existing ones;

2. parallel algorithms and programming techniques to support large scale and high-performance computing;

3. development of robust and efficient state-of-the-art numerical solvers suitable for handling a wide range of flow regimes, ranging from incompressible to hypersonic speed;

4. integration of multiple models to provide more accurate description of complex physical phenomena, including transport properties, reaction kinetics and statistical thermodynamics.

As far as the OO design is concerned, a number of structural solutions for creating reusable and extendible scientific software have been developed in collaboration with other team members [68, 131] and will be presented. We will propose suitable design patterns for providing a uniform way of encapsulating numerical algorithms for solving Partial Differential Equation (PDE), physical models and their interaction within multi-physics simulations. We will also present solutions for handling generic mesh data-structures with a transparent treatment of both serial and distributed data.

Given our intention to support complex and computational intensive simulations, we will describe the efficient parallel algorithms for mesh decomposition and parallel input/output that we have implemented and tested on large cluster architectures.
Since our final target is to deal with aerothermodynamic applications in conditions ranging from perfect non reactive gas to thermo-chemical equilibrium to full nonequilibrium, especially in the hypersonic regime, many physical models had to be integrated in COOLFluiD and they will all be described. With regard to this, some details will be left out of this thesis, since physical modeling was not the main focus of this work and since most of the chemistry, thermodynamics and transport algorithms were used as a black box, through the interface of Mutation [94, 116, 140]. The latter is a research library which has been developed at the VKI during the last decade and has been fully linked to COOLFluiD within this work, in order to make it usable in real-life CFD simulations.
Furthermore, we will focus our attention on numerical schemes which combine accurate shock capturing and robustness. The standard cell-centered FV is the best established method and usually the preferred choice for these applications, in combination with implicit time stepping techniques to accelerate the convergence. During the last decade, however, RD schemes have been emerging as a possible valid alternative for the computation of compressible flows on unstructured meshes. We will show that our flexible platform has allowed us to develop and compare both methodologies. In particular, in this thesis we will extend, for the first time in literature, the strictly conservative variant of RD schemes, aka CRD, to the simulation of viscous chemically reactive flows.

Part I

Design Solutions for Scientific Computing

Chapter 2

Object Oriented design solutions

Introduction

Designing a framework for generic PDE solvers is not a trivial task by itself, but additional complexity has come, in our case, from having planned to deal efficiently with arbitrary numerical algorithms (with potentially different parallel data structures) and multi-physics applications.
According to a simplified conceptual view, we can consider a numerical solver for PDE's as made by a limited number of independent building blocks:

- some mesh-related data, typically numerics-dependent, that represent a discretized view in terms of both geometry and solution of the computational domain;
- an equation set describing the physical phenomena to study, typically independent of numerics;
- a group of interacting numerical algorithms to solve the PDE's on the given discretized domain.

We can therefore identify three issues of fundamental importance for developing an environment for solving PDE's:

- storage and handling of mesh data;
- implementation of physical models;
- implementation of numerical methods.

A good OO design should be able to tackle these three aspects orthogonally, independently one from the other as much as possible, in order to minimize coupling and maximize the reusability of the single components. To this aim, a clear separation should be enforced between physics and numerics, the first being the description of the properties, constitutive relations and

quantities that characterize the equation system, the second providing the mathematical tools (algorithms, algebraic and differential operators) to discretize and solve those equations.
This section will start with a short introduction to basic concepts in object-oriented programming and design like inheritance, polymorphism and composition which will be reused heavily later on. After that, our attention will focus on analyzing each one of the three above-mentioned problems (2.2, 2.3, 2.4), and on presenting corresponding possibly effective solutions, as they were implemented in COOLFluiD. Some illustrative examples will also be given in Sec. 2.3.1 and 2.4.1 in order to show the concrete applicability of the proposed ideas.

2.1 Basic concepts in OO Programming

OO programming consists in a new conception of programming based on an accurate conceptual analysis of a given problem that leads to the identification of the objects involved, their factorization into classes, the definition of class interfaces and hierarchies and the establishment of key relationships among them. This, at start, requires considerably more effort than writing a flow chart and translate it into actual code, as procedural (aka functional) programming would normally require, but it's also considerably more powerful, because it allows software developers to go beyond the specific problem and create the basis for reusable code.
OO programming is usually called *programming to the interfaces*, which tells a lot about the effort to generalize and abstract as much as possible to determine what are the objects involved and how these objects interact through their interfaces.
Some key ingredients of the OO approach, as indicated in [33], are

- **data abstraction:** the logical properties of data types are separated from their implementation;

- **data encapsulation:** the physical representation of the data of an algorithm is surrounded, so that the user of the data doesn't see the implementation, but deals with the data only in terms of its logical picture, namely its abstraction;

- **information hiding:** the details of a function or data structure are hidden in order to control access to them;

- **polymorphism:** the ability to determine dynamically (at run-time) or statically (at compile time) which of several operations with the same name is appropriate.

2.1.1 C++ techniques

C++ has been the programming language chosen for the actual implementation of the COOLFluiD framework, since it provides a lot of attractive features and allows developers to use very effective techniques, at the cost, sometimes, of a quite complex syntax. Occasionally, some minor portability problems between one compiler and another can still be encountered, but this has become more and more rare since the standardization of the language. Another valid reason to choose C++ for implementing an OO platform nowadays is the large availability of multi-purpose libraries for scientific computing in C/C++ which are easily linkable, including the ones (LAM, MPICH, OpenMP) implementing Message Passing Interface (MPI) for parallel and distributing computing.
Some of the main OO techniques supported by C++ will now be briefly described, in order to help the reader familiarize with some terminology which will recur often in the rest of the chapter.

2.1.1.1 Class Inheritance

This is a mechanism that lets developers implement one class interface defining an object in terms of another existing one. When a *derived class* inherits from a *base class*, it includes the definition of all the data and the operations defined by the parent class. A class is said to be *abstract* if its only purpose is to define a common interface for its subclasses. This implies that all or some implementations of the operations are deferred to the derived classes, making the abstract class non instantiable. Classes that aren't abstract are called *concrete*. Fig. 2.1 shows the OMT (Object Modeling Technique) [48] notation for class inheritance, with a vertical line and a triangle pointing towards the base class.

2.1.1.2 Dynamic and static binding

When identical requests that may have different implementations are forwarded to the abstract interface of a base class, inheritance can be used to obtain the already mentioned polymorphism (see 2.1). As a consequence, the correct derived class will be selected to fulfill the request appropriately.

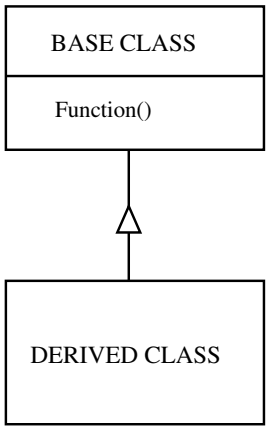

Figure 2.1: OMT diagram representing class inheritance.

When this selection occurs at run-time, the association is called *dynamic binding*, otherwise, when it occurs at compile time, it is called *static binding*.

2.1.1.3 Object Composition

Object composition is an alternative to class inheritance. It consists of assembling or *composing* objects to get more complex functionalities. The composition is defined dynamically at run-time through objects keeping references (or pointers) to other objects and forcing them to respect each others interfaces.
Each object of the composition can be considered as a *black box*, because no internal details (data, algorithms) of the objects are visible from outside the class. This allows to limit implementation dependencies and maintain encapsulation, leading to the so-called *black-box reuse* [48].
The OMT notation for composition is presented in Fig. 2.2. Composition can imply either *aggregation*, indicated by an arrow line starting with a diamond, or *acquaintance*, indicated by a plain arrow line. Aggregation requires that one object owns and/or is responsible for the other one, meaning that aggregatee and aggregator have identical lifetimes. Acquaintance implies that an object simply has knowledge of another one, without having ownership on it.

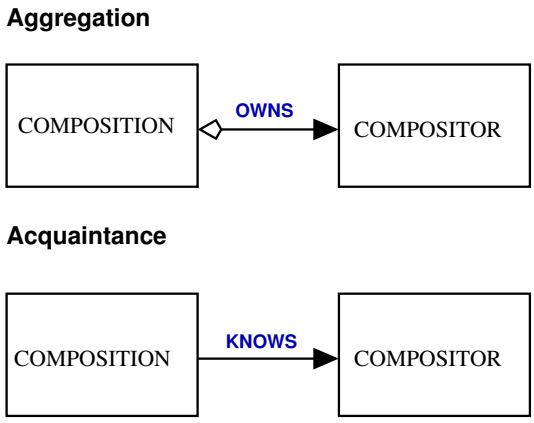

Figure 2.2: OMT diagram representing class compositions.

2.1.1.4 Templates

This technique allows developers to define a type (other name for class) without specifying all the other types it uses. The unspecified types are supplied as *parameters* at the point of use. As an example, we can consider a container (list, vector, set, etc.) that must handle different kind of elements (int, double, char, whatever fancy object) in the same way, and it's therefore parameterized with the element type.
Templates can help to create highly generic, efficient and reusable code. The most powerful example of application of this technique is the so called Standard Template Library (STL), that has been extensively used in the present work.
For further details on how to implement this techniques in practice, which is clearly out of the scope of this work, we particularly recommend to look at [149], [98], [160], [10].
For the interested reader, however, we provide in Appendix (Sec. A.1) an example of powerful use of template-based meta-programming for achieving an optimal implementation of symbolic array algebra. In the latter, we show how to enhance the state-of-the-art Fast Expression Template (FET) technique [57, 66] in order to make it usable within software frameworks with dynamically linked plug-ins, like COOLFluiD, and in order to allow the compiler to perform an optimization known as loop unrolling.

2.2 Mesh Data Handling and Storage

PDE-driven simulations typically involve a considerable amount of data related to the discretization of the computational domain, such as coordinates, state variables, geometric entities, etc., all grouped inside container objects (arrays, lists ...) on which numerical actions are performed. Additional complexity appears when computing on unstructured grids, as in the case of COOLFluiD, since different numerical methods can require the storage and usage of different kind of connectivity information. As a result, genericity in both quantity and types of data must be addressed.
Moreover, in a scalable and safe design of the data handling, framework components should be able to share data without violating encapsulation.
In a high performance environment, developers of new modules should be allowed to elect newly defined data types to be stored, handled efficiently, and shared in parallel communication. Ideally, local and distributed data should be treated uniformly from the developer's point of view, and support for different communication strategies (e.g. message passing and shared memory) should be offered.

2.2.1 Data Storage

As explained in [78, 80], in COOLFluiD, all massive data whose size scales with the problem complexity are encapsulated by a facade object `MeshData`, which, in particular, aggregates and offers safe access to a number of underlying instances of `DataStorage`.

```
1 template <class TAG> class DataStorage {};
```

Code Listing 2.1: Empty class DataStorage class definition

`DataStorage` is defined in C.L. 2.1 as a template empty class parameterized with a lightweight tag policy, `TAG`, which is meant to specify the actual parallel implementation or communication type, e.g. `LOCAL`, `MPI`, `PVM`, `SHM`:

```
  class LOCAL {}; // serial communication
2 class MPI   {}; // MPI-based parallel communication
  class PVM   {}; // PVM-based parallel communication
  class SHM   {}; // shared memory communication
```

Code Listing 2.2: Tag classes defining the communication models

As explained in [69], these tags do not appear explicitly in the code, but

they are aliased at compile time to more general concepts, such as `GLOBAL` for inter-communication or `REMOTE` for intra-communication, according to the available models and the user's preferences.

```
#ifdef HAVE_MPI
typedef MPI GLOBAL;
#elif HAVE_PVM
typedef PVM GLOBAL;
#else
typedef LOCAL GLOBAL;
#endif

#ifdef HAVE_SHM
typedef SHM REMOTE;
#else
typedef LOCAL REMOTE;
#endif
```

Code Listing 2.3: Macros definitions for GLOBAL and LOCAL communication models

In COOLFluiD, a parallel layer is defined with the task of concealing implementation details that rely upon specific parallel paradigms. To this aim, only the aliased tags are used outside the parallel layer, i.e. in the numerical modules of the framework.
`DataStorage` maintains a key role within the parallel layer, by providing a uniform, user-friendly interface to create and manage arrays of generic data types, that are meant to be shared among different numerical components. In particular, DataStorage offers the same interface to treat both local and distributed data uniformly, even though the actual implementation of its member functions depends on the chosen `TAG` class. In other words, every choice of a `TAG` class leads to a different instantiable explicit template specialization [160] of `DataStorage`, as clarified here after.

2.2.1.1 Local Data Storage

The class definition of `DataStorageLocal` is reported in C.L. 2.4: it provides functions to create, get and delete local data, which are stored in a `Evector` [32]. The latter offers an improved and more efficient implementation of `std::vector` [29, 149, 150], the basic container provided by STL. In particular, `Evector` supports back insertion/deletion of elements in constant time and on-the-fly memory allocation to accommodate new entries, without any overhead. `DataStorageLocal` behaves as an archive, where ar-

rays of data with a certain type and size are registered under a user-defined name in an associative container like a `std::map` [149] [150]. Pointers to the arrays in question are statically casted to `void*`, in order to let coexist different types in the same instance of `DataStorageLocal`. No built-in garbage collection or deletion policies based on some sort of reference counting are automatically provided by `DataStorageLocal`. This fact implies that data created with `createData()` must be consistently deleted with an explicit call to `deleteData()`: ad-hoc *setup* and *unsetup* `Commands` fulfill this task in each numerical module, as explained in Sec. 2.4.1.

```
// --- DataStorageLocal.hh --- //

template<> class DataStorage<LOCAL> {
public:
  // Creates and initializes a storage
  template <class T> DataHandle<LOCAL,T> createData
  (string name, int size, T init = T());

  // Deletes a storage and frees its memory
  // Returns the number of deleted storages
  template <class T> int deleteData(string name);

  // Gets an existing storage
  template <class T> DataHandle<LOCAL,T> getData
  (string name);

private:
  // map that stores the pointers to arrays of data
  std::map<string, void*> m_dataStorage;
};
```

Code Listing 2.4: Explicit specialization for LOCAL DataStorage

2.2.1.2 MPI Data Storage

The `DataStorage` associated to the `MPI` tag, namely `DataStorageMPI`, is defined in C.L. 2.5. Since `DataStorageMPI` requires some MPI specific assumptions, it provides its own implementation of the same interface of `DataStorageLocal`. The main difference between `DataStorageLocal` and this class is given by the presence of a MPI *communicator*, needed by `ParVectorMPI`, which is the container type for the data to be registered in `DataStorageMPI`.

```
// --- DataStorageMPI.hh --- //

template<> class DataStorage<MPI> {
public:
  // constructor accepting a MPI communicator
  DataStorage (MPI_Comm communicator);

  // Creates and initializes a storage
  template <class T> DataHandle<MPI,T> createData
  (string name, int size, T init = T());

  // Deletes a storage and frees its memory
  // Returns the number of deleted storages
  template <class T> int deleteData(string name);

  // Gets an existing storage
  template <class T> DataHandle<MPI,T> getData
  (string name);

private:
  // map that stores the pointers to arrays of data
  std::map<string, void*> m_dataStorage;
  MPI_Comm m_communicator; // communicator
};
```

Code Listing 2.5: Explicit specialization for MPI DataStorage

2.2.1.3 Data Handle

Once registered in either a local or MPI `DataStorage`, data can be accessed through a smart pointer, `DataHandle`, that ensures safety by preventing the data array from being accidentally deleted. It also provides inlinable accessor/mutator functions to the individual array entries and offers some additional functionalities. The class definition of `DataHandle` is presented in C.L. 2.6.

```
// --- DataHandle.hh --- //

  template <class TAG, class T> class DataHandle {};
```

Code Listing 2.6: Empty definition of DataHandle

Different partial template specializations of `DataHandle` exist, one for each tag class. As an example, the class definition for the `DataHandle` corresponding to the `MPI` tag is given in 2.7.
`DataHandleMPI` declares functionalities that deal with the synchronization

```
// --- DataHandleMPI.hh --- //

template <class T> class DataHandle<MPI, T> :
    public DataHandleInternal <ParVectorMPI,T> {
public:
  // constructors

  void buildSyncMap(); // build mapping for synchronization
  void beginSync(); // begin synchronization
  void endSync();   // end synchronization

  int addGhostPoint (int globalID); // add ghost point
  int addLocalPoint (int globalID); // add local point
  int getGlobalSize() const; // get size of underlying array
  void reserve(int n); // reserve size capacity
};
```

Code Listing 2.7: DataHandleMPI interface

and handling of data belonging to the overlap region. The actual synchronization and communication is delegated to the acquainted parallel vector, `ParVectorMPI`, in which all MPI calls are encapsulated. The same functions must be provided also by `DataHandleLocal` (with an empty implementation) in order to allow the whole code to compile even without having MPI, i.e. when the `GLOBAL` tag is aliased to `LOCAL`. For sake of completeness, we report in C.L. 2.8 the definition of `DataHandleInternal`, from which all `DataHandles` derive. In particular, the overloading of the subscripting `operator[]` allows to directly access array entries through the `DataHandle` interface.

2.2.2 Topological Region Sets

The computational domain can be considered as composed by topologically different regions which correspond to the `TopologicalRegionSet` concept in COOLFluiD. Each TopologicalRegionSet (TRS) consists of a list of `TopologicalRegions`, i.e. subsets of the domain ideally linked with a particular Computer Aided Design (CAD) representation. Each set of boundary surfaces on which a certain boundary condition has to be enforced can be interpreted as a different TRS, while the whole boundary can be broken into smaller patches, TopologicalRegion (TR), according, for instance, to the surface definition in the CAD model. If we consider, as an example, the computational domain built around an airplane, FarField, Wing, Nacelle, Fuselage can all be considered as different TRS's. One could alternatively

```
// --- DataHandleInternal.hh --- //

template<template <class T> class STYPE, class T>
class DataHandleInternal {
public:
  typedef STYPE<T> StorageType;

  // constructor
  DataHandleInternal(StorageType *const ptr) : m_ptr(ptr){}
  // copy constructors, assignment operator ...

  // overloading of subscripting operator
  T& operator[] (int idx) {return (*m_ptr)[idx];}

  // overloading of operator ()
  T& operator() (int i, int j, int stride)
  {return (*this)[i*stride+j];}

  // ...
protected:
  StorageType* m_ptr; // pointer to the array to be handled
};
```

Code Listing 2.8: DataHandleInternal interface

choose to pack Wing, Nacelle and Fuselage together to form a unique continuous TRS, called Airplane, consisting of three TR's.
Because of consistency and conveniency, not only the boundary but also the interior of the domain is assigned to one or more TRS's, according to the needs of the chosen numerical methods (space/time discretizations, mesh adaptation, mesh movement, coupling algorithms, etc.).
The latter typically are applied on geometric entities (cells, faces, edges), which represent the basic entities in the topological dicretization of the domain. A `GeometricEntity` is defined by an algorithm-dependent agglomeration of degrees of freedom, both in geometric (`Nodes` or coordinates) and solution space (`States` or solution vectors). Both `Nodes` and `States` are provided with global (inter-process) and local (intra-process) IDs, which are used to build connectivity information. As a result, each TRS's (or TR's) can be logically viewed as set (or subset) of GeometricEntity (GE) in which each portion of the domain can be decomposed for computational purposes.
A TRS is a *Proxy* object [48] that controls the access to:

- the local and global IDs of all the GE's belonging to the TRS;

- the local IDs of all nodes and states referenced by local GE's;
- the geo-type information (i.e. the unique IDs corresponding to the shape and polynomial order of each GE);
- the geo-to-node and geo-to-state connectivities;

as shown in the diagram in Fig.2.3. In order to minimize the memory requirements, all these data are stored sequentially in unidimensional arrays (but logically bidimensional) which can be addressed by the client code exclusively through the TRS or TR interfaces, with the latter offering only a partial view of the whole data.

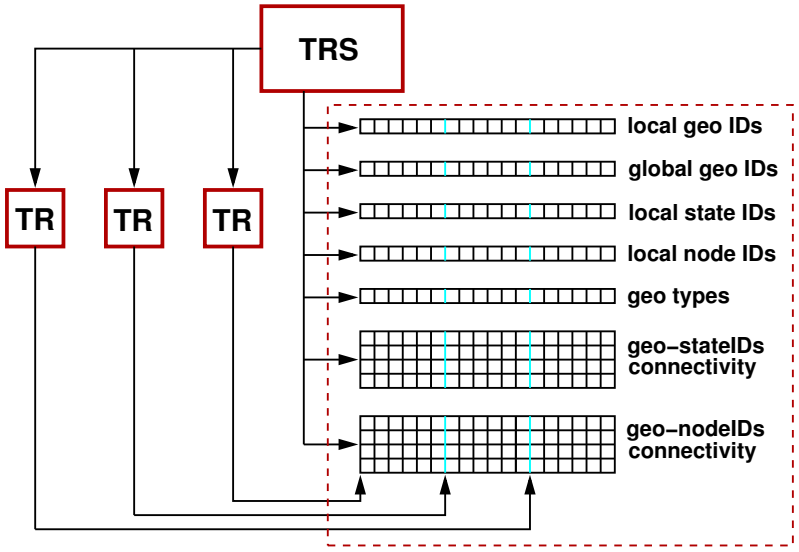

Figure 2.3: Diagram showing the array-based data-structure associated to a TRS and the underlying TR.

2.2.2.1 On-the-fly creation of Geometric Entities

As mentioned in the previous paragraph, numerical algorithms are applied on cells, faces, edges, all corresponding to GE instances in our design. A GE is a locally/globally indexed object which is composed by `States`, `Nodes`, `ShapeFunctions` for the solution and geometric interpolation and it can keep reference to other neighbor GE's, according to the computational stencil

needed by the numerical method. A simplified interface for GE is shown in C.L. 2.9.

```
// --- GeometricEntity.hh --- //

class GeometricEntity {
  public:
  // constructor and destructor, utility functions ...

  // accessor functions ...
  SafePtr<vector<State*> > getStates() const;
  SafePtr<vector<Node*> > getNodes() const;
  SafePtr<vector<GeometricEntity*> > getNeighborGeo() const;

  // mutator functions
  void setState(int is, State* state);
  void setNode(int in, Node* node);
  void setNeighborGeo(int ig, GeometricEntity* neighborGeos);

protected: //data
  int m_geoID; // local ID of this GeometricEntity
  std::vector<State*> m_states; // state list
  std::vector<Node*> m_nodes;   // node list
  std::vector<GeometricEntity*> m_neighbors; // neighbor GEs
  ShapeFunction* m_solSF; // solution shape function
  ShapeFunction* m_geoSF; // geometric shape function
};
```

Code Listing 2.9: GeometricEntity interface

As one can easily notice, a `GeometricEntity` is a relatively lightweight object, but it would be prohibitely expensive to carry in memory all the cells and/or faces and/or edges corresponding to the given mesh and numerical algorithm along the whole simulation. It is however extremely convenient to have such an object that encapsulates all the needed local connectivity, shape function-related, stencil-related data and that offers them ready-to-use.

This lead us to the idea of applying a *Builder* pattern [48] to separate the construction of GE's from their representation, starting from a preallocated memory pool. The pattern is represented in Fig. 2.4.

We define a generic interface for `GeoEntityBuilder`, which is a template class parameterized with the actual builder, called `GEOBUILDER` in C.L. 2.10. `GeoEntityBuilder` just forwards actions to the aggregated `GEOBUILDER`, which is imposed by the chosen numerical algorithm.

Each possible choice for the parameter `GEOBUILDER`, corresponds to a dif-

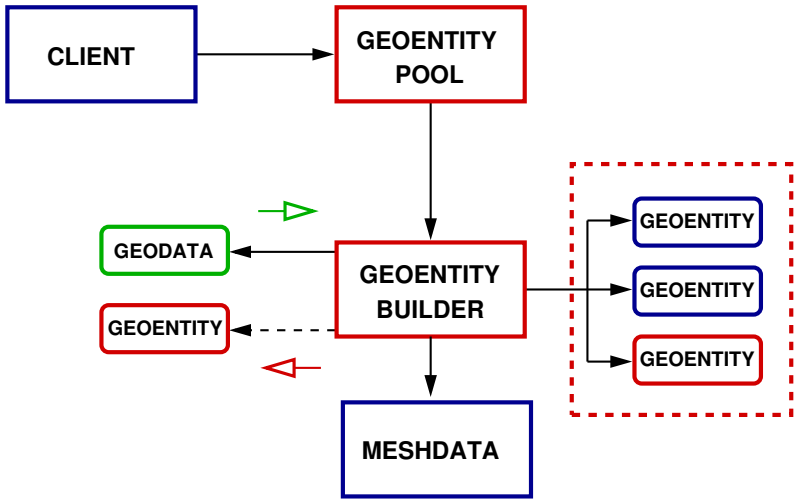

Figure 2.4: Builder pattern for creating GeometricEntities.

```
// --- GeoEntityBuilder.hh --- //

template <class GEOBUILDER>
class GeoEntityBuilder {
public:
  // constructor, destructor

  // allocate the prototype GEs and set up GEOBUILDER data
  void setup() {m_builder.setup();}

  // get the input data for the GEOBUILDER
  typename GEOBUILDER::GeoData& getDataGE()
  {return m_builder.getDataGE();}

  // build the GE corresponding to the given input data
  GeometricEntity* buildGE() {return m_builder.buildGE();}

  // release the built GEs to prepare the next creation
  void releaseGE() {m_builder.releaseGE();}

private: //data
  GEOBUILDER m_builder; // actual GE builder
};
```

Code Listing 2.10: GeoEntityBuilder interface

```
// --- FEMCellBuilder.hh --- //

class FEMCellBuilder {
public:

  // This nested struct groups the data needed by this builder
  struct GeoData {
    SafePtr<TopologicalRegionSet> trs; // pointer to TRS
    int idx; // geo index in TRS
  };

  // allocate the prototype GEs and set up GEOBUILDER data
  void setup();

  // get the input data for the FEMBuilder
  FEMBuilder::GeoData& getDataGE() {return m_data;}

  // build the GE corresponding to the given input data
  GeometricEntity* buildGE();

  // release the built GEs to prepare the next creation
  void releaseGE();

private:
  FEMBuilder::GeoData m_data; // data of this builder
  DataHandle<Node*> m_nodes;  // handle to the Nodes storage
  DataHandle<State*> m_states;// handle to the States storage
  std::vector<GeometricEntity*> m_poolGE; // pool of
      preallocated GEs
};
```

Code Listing 2.11: FEMBuilder interface

ferent instance of `GeoEntityBuilder`. As an example, C.L. 2.11 shows the class definition of a simple `FEMCellBuilder`. The latter, in particular, provides a nested struct `GeoData`, holding the numerics-dependent input data for the creation of a GE.
During the setup phase of the simulation, a small number of GE objects, corresponding to all the entities belonging to the computational stencil required by the actual application, is created and stored in an array. While applying numerical algorithms, after the `GeoData` are set by the statically bound client code, the builder is asked to assemble a ready-to-use GE, where all the data have been fetched out from the corresponding TRS and `DataHandles`. The builder operates on the prototype GE's that have been already previously allocated in memory, but it changes their content (data pointers) according

to the current request. The GE creational algorithm can be as complex as one needs, but all its complexity and data dependencies are hidden to the client code that, in our example, just reduces to C.L. 2.12.

```
1
  SafePtr<GeometricEntityPool<FEMCellBuilder> > builder;
  builder.setup();
4 ...

  FEMCellBuilder::GeoData& geoData = builder.getDataGE();
7 geoData.trs = getCurrentTRS();
  geoData.idx = getCurrentGeoIDInTRS();

10 builder.buildGE();
  ...
  builder.releaseGE();
```

Code Listing 2.12: FEMBuilder interface

This separation between the GE construction and the GE data object itself has allowed us to easily deal with the most diverse space discretizations and associated data structures such as Finite Element Method (FEM), cell centered FV, Spectral FV, Spectral FD, DG, each one providing an ad-hoc definition and implementation of a concrete GE builder.
`GeoEntityBuilder` offers an example of static polymorphism, where, instead of defining an abstract interface and implement its virtual member functions in the derived classes, actions are delegated by `GeoEntityBuilder` to an aggregated interface which is statically bound to the client code. This choice offers superior performance (no virtual calls on some inlinable functions in tight loops) and it reflects a logical dependence of the concrete numerical method on the concrete way of creating suitable GE's for it.

Memory optimization. On-the-fly creation of GE's has allowed us to heavily optimize the code in terms of memory, compared to a previous approach where all GE's were allocated and actually stored in the TRS's. This improvement made memory usage competitive against procedural (as opposite to object-oriented) highly optimized numerical codes, such as Elsyca's PlatingMaster, an industrial FEM solver (written in C) for electrochemical applications.

2.3 Implementation of Physical Models

Scientists and researchers that work in the field of Fluid Dynamics deal regularly with complex systems of convection-diffusion-reaction PDE's that can be formulated as

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \mathbf{F^c} = \nabla \cdot \mathbf{F^d} + \mathbf{S} \tag{2.1}$$

where $\mathbf{U}$ (state vector of unknown variables), F^c (convective fluxes), F^d (diffusive fluxes), $\mathbf{S}$ (source or reaction term) depend on the chosen physical model. The latter can be seen as a composition of entities, e.g., transport coefficients, thermodynamic properties and quantities that contribute to define the mathematical description of some physical phenomena. It can be noticed that one can look at the same physical model through different formulations of the corresponding equations, which involve the use of different sets of variables, transformations, or other adaptations tailored towards a specific numerical method.
Moreover, in a framework for solving PDE's which supports different numerical techniques and run-time loading of components, the modules where algorithms are implemented should be as independent as possible from the modules dealing with the physics description. Therefore, when new physical models or numerical components are integrated, the need for modifications on the kernel, where most of the key abstractions are defined, should be avoided. Ideally, it should be possible to build more complex models by simply extending and reusing available ones, or solving the same equations by means of different numerical techniques, without requiring changes in the existing abstract interfaces and without establishing a high level direct coupling between physics and algorithms. Widely used OO CFD platforms like [6] or [44] lack these last features, since the implementation of the physics is directly tangled to the numerical solver, in particular to the space discretization. This is surely a reasonable down-to-earth solution, it is probably efficient, but it does not promote enough reusability and interchangeability between physics and numerics. The `Perspective` pattern, that will now be described, proposes a structural way for overcoming the above mentioned pitfalls and for facilitating dynamical incremental changes and code reuse.

2.3.1 Perspective Pattern

Trying to define a single abstract interface for a generic physical model would be quite a demanding task and it would easily lead to a non maintainable

solution, especially if we keep in mind our concern with run-time flexibility. In COOLFluiD, this hypothetical hard-to-define interface is therefore broken into a limited granularity of independent *Perspective* objects [78, 80], each one offering a different view of the same physics, according to the needs of possibly different numerical algorithms. To make an example, convective, diffusive, inertial, reaction or source terms of the equations can all be Perspectives with a certain abstract interface. Moreover, if one implements a new numerical scheme that needs to access an available physical model, but that requires something not foreseen a priori and, therefore, not offered by the available interfaces, a new Perspective can be created, without affecting the existing ones.
Figure 2.5 shows the OMT class diagram of a Perspective pattern applied to a generic physical model. The base `PhysicalModel` defines a very general and narrow abstract interface. `ConcreteModel` derives from `PhysicalModel`, implements the virtual methods of the parent class and defines another interface to which the concrete Perspective objects (and only those) are statically bound. This other interface offers data and functionalities which are typical of a certain physics, but invariant to all its possible Perspectives.

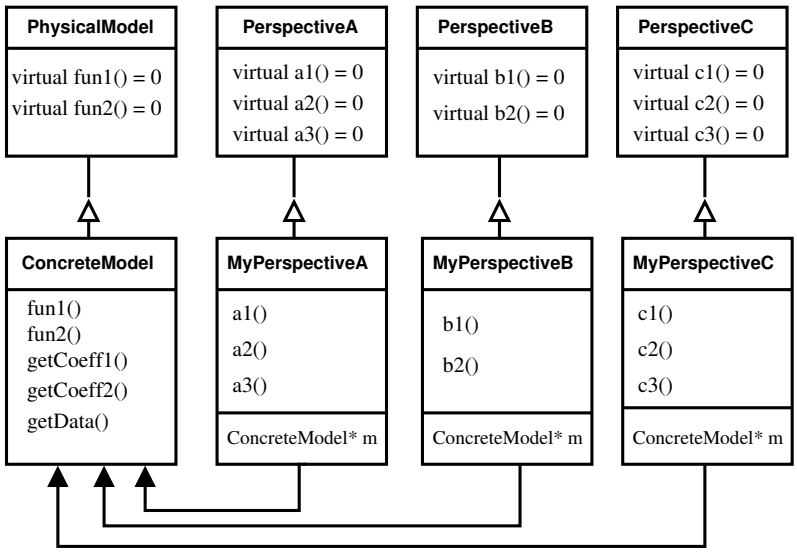

Figure 2.5: Perspective pattern applied to a Physical Model.

The resulting pattern lets the numerical client code make use of the physi-

cal model through an abstract layer, dynamically enlargeable if required, given by a number of Perspective objects, while all their collaborations with the `ConcreteModel` are completely hidden. The pattern reflects the composition-based *Adapter* described in [48], but with a single shared *Adaptee* object, i.e. `ConcreteModel`, and multiple abstract *Targets* (called Perspectives here), each one with a number of derived classes (*Adapters*). The fact that several objects may be defined to describe the same physics may look like a disadvantage, but, in our experience, improves reusability, allows developers to easily decouple physics from numerics and gives better support to a run-time plug-in policy.
Unlike in other OO CFD platforms [28, 44] where no neat distinction between numerical algorithms and physical description is searched and where a close form of reliance (inheritance) binds the equation model and the scheme, in COOLFluiD, physics is generally completely independent and unaware of numerical methods (space and time discretizations, linear system solving, etc.). However, a design that is based on Perspective objects does not even prevent from having numerical objects, such as *Strategies* or *Commands* in 2.4.1, with static binding to the actual physics, e.g. in the case of some special schemes or boundary conditions. In the latter case, the use of Perspectives can still help to limit dependencies, improve reusability and avoid excessive sub-classing.
Some code samples are now presented in order to show the concrete applicability of the described pattern for handling complex physical models.

2.3.1.1 Physical Model Object

At first, the interface of the base `PhysicalModel` is defined in C.L. 2.13.
A polymorphic `BaseTerm` object, as defined in C.L. 2.14, and a corresponding enumerated variable are assigned to each independent physical term of the equation (convection, diffusion, reaction, dispersion, inertia, etc.) and they are registered in an associative container, such as a `std::map`. `BaseTerm` manages a number of unidimensional arrays of floats (`RealVector`), where some physics-dependent data (thermodynamic quantities, transport coefficients, parameters useful for calculating fluxes, eigenvectors, eigenvalues etc.) are stored and continuously recomputed during the simulation.
In order not to affect the total memory requirements of the computation, the maximum number of these arrays is determined by the numerical method and scales with the number of degrees of freedom in one or more geometric entities (cells, faces, etc.), typically depending on the required computational stencil. To make an example, for a cell-vertex based algorithm

```
// --- PhysicalModel.hh --- //

class PhysicalModel {
public:
  // constructor, virtual destructor ...

  // enumerated variables used to define equation term types
  enum TermID {CONVECTION, DIFFUSION, REACTION, DISPERSION};

  virtual void setup() = 0;                //set up private data
  virtual int getDimension() const = 0;    //geometric dimension
  virtual int getNbEquations() const = 0; //number of equations

  // accessor-mutator function returning the BaseTerm
  // corresponding to the given TermID
  SafePtr<BaseTerm> getTerm(TermID t) {return m_tMap.find(t);}

protected:
  // registers the TermID and a pointer to the
  // corresponding polymorphic BaseTerm object
  void registerTerm(TermID t, BaseTerm* bt)
      {m_tMap.insert(t,bt);}
private:
  // mapping between TermID and a polymorphic physical term
  Map<TermID, SafePtr<BaseTerm> > m_tMap;
};
```

Code Listing 2.13: PhysicalModel interface

(FEM, RD, etc.), the total number of physical data arrays which are stored in `BaseTerm` could be the maximum number of quadrature points in a cell. The size and the content of these arrays will be defined by the classes deriving from `BaseTerm`.
A template compositor object is defined for each possible combination of physical equation terms (`Convection`, `Diffusion`, `ConvectionDiffusion`, etc.). It derives from `PhysicalModel` and aggregates generic policies [10], i.e. subclasses of `BaseTerms`. In C.L. 2.15 we present the class definition of a `ConvectionDiffusion` compositor, while its constructor implementation is shown in C.L 2.16. The equation terms aggregated by the compositor are registered in the parent class and each one of them is made accessible polymorphically through the `getTerm()` method defined in `PhysicalModel`.

Example: Navier-Stokes Model. We consider the case of a Navier-Stokes model: an overview of its class diagram is provided in Fig. 2.6.
The class definitions for the convective `EulerTerm` and the diffusive `NSTerm`

```
// --- BaseTerm.hh --- //

class BaseTerm {
public:
  // constructor, virtual destructor ...

  // sets the maximum number of physical data arrays:
  // this size must be set by the numerical solver
  // because it depends on the computational stencil
  void setNbDataArrays(int maxSize);

  // get the size of each array of temporary physical data
  virtual int getDataSize() const = 0;

  // finds the array of data  corresponding to the given
  // degree of freedom (state vector)
  SafePtr<RealVector> getPhysicalData(State* state);
};
```

Code Listing 2.14: BaseTerm interface

```
// --- ConvectionDiffusion.hh --- //

template <class CT, class DT>
class ConvectionDiffusion : public PhysicalModel {
public:
  // constructor, virtual destructor,
  // overridden parent class pure virtual methods
protected:
  std::auto_ptr<CT> m_convTerm; // convective term
  std::auto_ptr<DT> m_diffTerm; // diffusive term
};
```

Code Listing 2.15: ConvectionDiffusion interface

```
template <class CT, class DT>
ConvectionDiffusion<CT,DT>::ConvectionDiffusion(string name) :
 PhysicalModel(name),
 m_convTerm(new CT(CT::getTermName())),
 m_diffTerm(new DT(DT::getTermName()))
{
  // register all the equation terms and their TypeIDs
  registerTerm(PhysicalModel::CONVECTION, m_convTerm.get());
  registerTerm(PhysicalModel::DIFFUSION, m_diffTerm.get());
}
```

Code Listing 2.16: ConvectionDiffusion constructor implementation

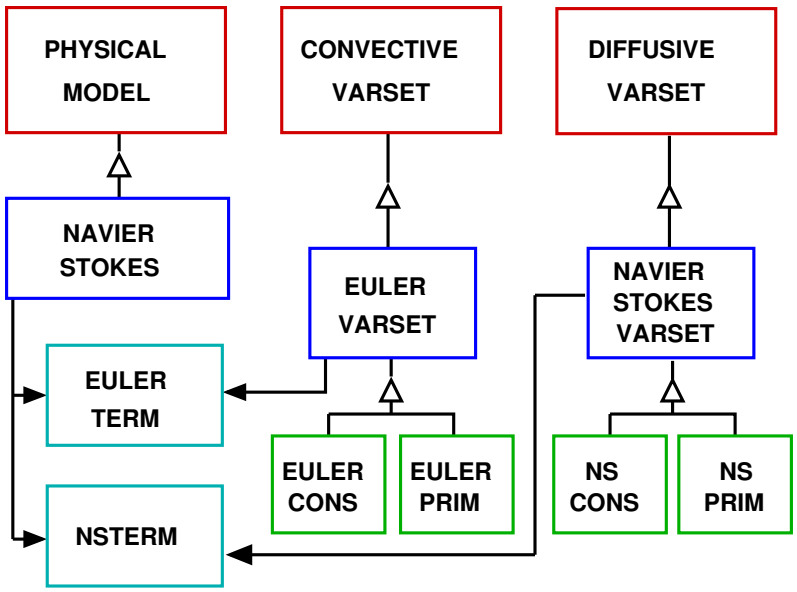

Figure 2.6: Perspective pattern applied to a Navier-Stokes model.

are shown in C.L 2.17 and 2.18.

```
// --- EulerTerm.hh --- //

class EulerTerm : public BaseTerm {
public:
  // enumerated variables: density, pressure, enthalpy, etc.
  enum {RHO=0,P=1,H=2,E=3,A=4,T=5,V=6,VX=7,VY=8,VZ=9};

  // number of the enumerated variables
  virtual int getDataSize() const {return 10;}

  CFreal getMachInf() const;  // freestream mach
  CFreal getTempRef() const;  // freestream temperature
  CFreal getPressRef() const; // freestream pressure
  static string getTermName() {return "Euler";}

  SafePtr<TDLibrary> getThermodynamicLibrary() const;
private:
  ... // member data
};
```

Code Listing 2.17: EulerTerm interface

```
// --- NSTerm.hh --- //

class NSTerm : public BaseTerm {
public:
  // dynamic viscosity, thermal conductivity
  enum {MU=0, LAMBDA=1};

  // number of the enumerated variables
  virtual int getDataSize() const {return 2;}

  CFreal getPrandtl() const;     // Prandtl number
  CFreal getReynoldsInf() const; // Reynolds number
  static string getTermName() {return "NS";}

  // get library to compute transport properties
  SafePtr<TPLibrary> getTransportLibrary() const;
private:
  ... // member data
};
```

Code Listing 2.18: NSTerm interface

Besides providing accessors methods for useful parameters and libraries computing physical quantities (i.e. thermodynamics or transport properties), `EulerTerm` and `NSTerm` define the size and, by means of enumerated variables, the entries for the arrays of physical data that are stored in `BaseTerm` and that have been previously introduced. An instantiable `NavierStokesModel`, i.e. what is called `ConcreteModel` in the Perspective pattern, can now be defined as in C.L. 2.19.

```
// --- NavierStokesModel.hh --- //

class NavierStokesModel : public ConvectionDiffusion
                          <EulerTerm, NSTerm> {
public:
  // constructor, virtual destructor
  virtual int getDimension() const;   //geometric dimension
  virtual int getNbEquations() const; //number of equations
  virtual int setup() const; //set up data
};
```

Code Listing 2.19: NavierStokesModel interface

`NavierStokesModel` can exploit the knowledge of both its concrete convective and diffusive terms to compute, for instance, adimensionalizing coefficients.

2.3.1.2 Variable Sets

Numerical algorithms are not direct clients of the `ConcreteModel`. As explained previously, the latter is used through a layer of polymorphic Perspective objects, whose collaborations are statically bound to the ConcreteModel and therefore potentially very efficient. A variable set, (`VarSet`), is a Perspective that decouples the usage of a physical model from the knowledge of the used variables. This, for instance, allows the client code to solve equations formulated in conservative variables, to perform intermediate algorithmic steps and to update in other ones (primitive, characteristic, etc.). In order to deal with a system of convection-diffusion equations, two variable sets classes are defined: `ConvectiveVarSet` and `DiffusiveVarSet`, respectively in C.L. 2.20 and 2.21.

```
// --- ConvectiveVarSet.hh --- //

class ConvectiveVarSet {
public:
  // constructor, virtual destructor, etc.

  virtual void setup()=0; //set up private data

  // set physical data starting from a state vector
  virtual void setPhysicalData
           (const State& state, RealVector& data)=0;

  // set state vector starting from physical data
  virtual void setFromPhysicalData
           (const RealVector& data, State& state)=0;

  virtual void setJacobians(...)=0;   //set jacobians matrices
  virtual void splitJacob(...)=0;     //split jacobian
  virtual void computeEigenSystem(...)=0; //set
      eigenvalues/vectors
  virtual void computeEigenValues(...)=0; //set eigenvalues
  virtual void computeFlux(...)=0;//compute the convective flux
};
```

Code Listing 2.20: ConvectiveVarSet interface

Among all the methods declared in the above two classes, particular attention is due to `setPhysicalData`() and its dual `setFromPhysicalData`(): the former starts from a given state vector (unknown variables) and sets the physical dependent data, whose entry pattern is defined by the the subclasses of `BaseTerm`; the latter does the opposite. In the case of the `EulerTerm`, for

```
// --- DiffusiveVarSet.hh --- //

class DiffusiveVarSet {
public:
  // constructor, virtual destructor, etc.

  virtual void setup()=0; //set up private data

  // set physical data starting from a state vector
  virtual void setPhysicalData
    (const State& state, RealVector& data)=0;

  virtual void computeFlux(...)=0; //compute the diffusive flux

  // set the variables whose gradients appear in the diffusive
  // fluxes, starting from the given State vectors
  virtual void setGradientVars(vector<RealVector*>& states,
       RealMatrix& values, int stateSize) = 0;
};
```

Code Listing 2.21: DiffusiveVarSet interface

instance, if we assume that a `State` $\mathbf{P}$ holds primitive variables (pressure, velocity components, temperature)

$$\mathbf{P} = [\mathrm{p}, \mathrm{v}_x, \mathrm{v}_y, \mathrm{v}_z, \mathrm{T}] \tag{2.2}$$

`setPhysicalData`() will compute an array $\mathbf{W}$ of physical quantities (density, pressure, total enthalpy, total energy, sound speed, temperature, velocity module and components):

$$\mathbf{W} = [\rho, \mathrm{p}, \mathrm{H}, \mathrm{E}, \mathrm{a}, \mathrm{T}, \mathrm{V}, \mathrm{v}_x, \mathrm{v}_y, \mathrm{v}_z] \tag{2.3}$$

from $\mathbf{P}$ by means of thermodynamic relations. $\mathbf{W}$ will be then reused to compute eigenvalues, fluxes etc. A key point for the flexibility of the design is that each `VarSet` has acquaintance of the physical equation terms, but not of the resulting `ConcreteModel`: this allows the developer to compose progressively more complex models with full reuse of the individual equation terms.

Example: Navier-Stokes VarSets. As shown schematically in Fig 2.6 and, in actual code, in C.L. 2.22 and 2.23, `EulerVarSet` and `NavierStokesVarSet` inherit the interface of the corresponding parent classes.
Some functionalities associated to a `VarSet` are independent on the variables in which one needs to work: advective fluxes, for instance, can always

```
// --- EulerVarSet.hh --- //

class EulerVarSet : public ConvectiveVarSet {
public:
  // constructor, virtual destructor,
  // overridden parent virtual methods
protected:
  SafePtr<EulerTerm> m_model;
};
```

Code Listing 2.22: EulerVarSet interface

```
// --- NavierStokesVarSet.hh --- //

class NavierStokesVarSet : public DiffusiveVarSet {
public:
  // constructor, virtual destructor,
  // overridden parent virtual methods
protected:
  SafePtr<NSTerm> m_model;
};
```

Code Listing 2.23: NavierStokesVarSet interface

be computed once that physical data like pressure, enthalpy, velocity are known, because their formulation is always the same due to the conservation property.
However, as a counter example, the Euler equations can be written in conservative, symmetrizing, entropy, characteristic (...) variables and each one of this formulation defines different jacobian matrices for the convective fluxes. The `VarSet` abstraction is particularly useful in these more complex cases, since we can let variable-dependent subclasses of `EulerVarSet` implement the jacobian matrices, according to the chosen formulation. Therefore you can have `EulerCons`(conservative), `EulerPrim`(primitive), `EulerChar` (characteristic), etc.
It is helpful to have derived classes for the `NavierStokesVarSet` too, since, for instance, the transport properties can be computed differently in chemically reactive or non reactive flows, in Local Thermodynamic Equilibrium (LTE) or in Thermo-Chemical Nonequilibrium (TCNEQ).
The method `computeFlux()` that calculates the diffusive flux is a template method [48] in `NavierStokesVarSet` and delegates the computation of the transport properties to subclasses, which have more specific knowledge. In the case of the "basic" Navier-Stokes model, as indicated in Fig. 2.6, we can

have `NavierStokesCons` for conservative variables and `NavierStokesPrim` for primitive variables.

2.3.1.3 MultiScalarTerm and MultiScalarVarSet

The Perspective pattern is especially designed to deal with cases where additional sets of equations must be attached to an existing equation system. In all these situations, a general treatment is applied, based on the combined use of `MultiScalarTerm` and `MultiScalarVarSet`. The former implements a generic extension to an existing convective physical term (e.g. `EulerTerm`), allowing to add an arbitrary number of subsets of N_l physical data (scalar variables), each one corresponding to the w_l quantity in a generic conservation equation written in the form:

$$\frac{\partial \rho w_l}{\partial t} + \nabla \cdot (\rho w_l \mathbf{v}) + \cdots = 0, \quad l = 1, \ldots, N_l \tag{2.4}$$

```
// --- MultiScalarTerm.hh --- //

template <typename BASE>
class MultiScalarTerm : public BASE {
public:
  // constructor, virtual destructor, setup functions

  // number of the enumerated variables
  virtual int getDataSize() const
  {return BASE::getDataSize() + m_nbScalarVars.sum();}

  // start position of scalar vars data of subset i
  int getFirstScalarVar(int i) const {return m_firstVars[i];}

  // number of scalar vars data of subset i
  int getNbScalarVars(int i) const {return m_nbScalarVars[i];}

  // number of subsets of scalar vars
  int getNbScalarVarSets() const {return m_nbScalarVars.size();}
private:
  std::valarray<int> m_nbScalarVars;// number of scalar vars
  std::valarray<int> m_firstVars; // ID of the first vars
};
```

Code Listing 2.24: MultiScalarTerm interface

To make a concrete example, in the case of a thermo-chemical non equilibrium model, 2 subsets are needed: one corresponds to the N_s continuity

equations for the chemical species, the other one corresponds to the N_v conservation equations for the vibrational energy of molecules. The interface of `MultiScalarTerm` is shown in C.L 2.24. Analogously, `MultiScalarVarSet`, whose interface is shown in C.L. 2.25 allows to treat the convective part of additional equations like 2.4 in concrete `ConvectiveVarSets`. This allows the developer to incrementally build more and more complex physical models, where new equations can easily be added.

```
// --- MultiScalarVarSet.hh --- //

template <class BASEVS>
class MultiScalarVarSet : public BASEVS {
public:
  typedef MultiScalarTerm<typename BASEVS::PTERM> PTERM;

  // constructor, virtual destructor

  // return the convective term
  SafePtr<PTERM> getModel() const{return m_mScalarModel;}

  // get some data corresponding to the subset of equations
      related with
  // this variable set. The most concrete ConvectiveVarSet must
      set
  // this data
  static std::vector<EquationSetData>& getEqSetData() {
    static std::vector<Framework::EquationSetData> eqSetData;
    return eqSetData;
  }

protected:
   // add an equation data with the specified number of
       equations
   void addEqSetData(int nbEqs);

   // compute eigenvalues and convective fluxes for scalar
       variables
   virtual void computeEigenValues(...);
   virtual void computeFlux(...);
private:
  SafePtr<PTERM> m_mScalarModel; // convective term
};
```

Code Listing 2.25: MultiScalarVarSet interface

In particular, one `EquationSetData` object is associated to each equation subset in order to keep knowledge of the corresponding subset ID and provide

the list of all the variables IDs belonging to that subset.

2.3.1.4 Variable Transformers

Another polymorphic Perspective is the `VariableTransformer`. This allows the client code to apply matrix similarity transformations with the form $\partial\mathbf{U}/\partial\mathbf{V}$ between variables or to compute one set of variables from another $\mathbf{U}(\mathbf{V})$ analytically (e.g. Euler primitive from Euler conservative and viceversa).
The combined use of `VarSets` and `VariableTransformers` gives the freedom to use different variables to update the solution, compute the residual, linearize jacobians, distribute residuals, without requiring any modification in the client numerical algorithms that work with dynamically bound Perspective objects, completely unaware of the actual physics.
The support for variable manipulation and variable independent algorithms which is offered by COOLFluiD represents an important feature that none of other well known CFD platforms such as [6] or [44] have.

2.4 Implementation of Numerical Methods

The solution of PDE's requires the implementation of different numerical techniques that deal with time discretization, space discretization, linear system solving, mesh adaptation algorithms, error estimation, mesh generation, etc.
In a typical OO design, e.g. like the ones proposed by [28], [43], [104], or [134], each one of these simulation steps is enclosed in separate object (or polymorphic hierarchies of objects) and different patterns are applied to make them all stick together and interact.
It would be advantageous to have just a single pattern, extraordinarily reusable, that allows the developer to encapsulate different numerical methods uniformly, and lets him/her focus more on the algorithm itself rather than on its already well-defined surrounding. This pattern should ease the integration of new algorithms, but also the implementation of different versions or parts of the same algorithm, while looking for optimal solutions and/or tuning for run-time performance.
One of the main achievements of COOLFluiD, as opposed to other similar OO environments, lies in having developed a uniform high level structural solution, potentially able to encapsulate and tackle efficiently, we believe, all possible numerical algorithms: the compound Method-Command-Strategy

(MCS)pattern.

2.4.1 Method-Command-Strategy Pattern

The MCS pattern [78, 80, 132] provides a uniform way to implement numerical algorithms and to compose them according to the specific needs.

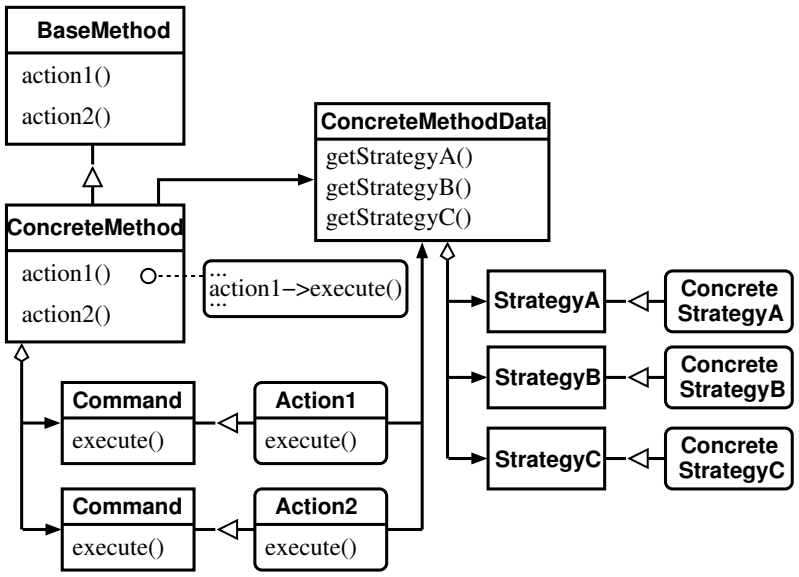

Figure 2.7: OMT diagram of the Method-Command-Strategy pattern.

As sketched in Fig. 2.7, `BaseMethod` defines an abstract interface for a specific type of algorithm (e.g. LinearSystemSolver, ConvergenceMethod, MeshCreator, SpaceMethod, ErrorEstimator ...). `ConcreteMethod` implements the virtual functions of the corresponding parent class by delegating tasks to ad-hoc `Commands`, see [10] [48], that share `ConcreteMethodData`, a tuple aggregating multiple polymorphic receivers (`Strategies`) [48].
Three levels of abstraction and flexibility can be identified in this pattern: `BaseMethod`, `Command` and `Strategies` can all be overridden, allowing one to implement the same task, at the corresponding level, in different ways. This kind of behavioral modularity allows the developers to easily re-implement or tune components (`Methods`, `Commands` or `Strategies`), and gives them the freedom to move code from one layer to another, according to convenience, taste or profiling-driven indications. The fast-path code, critical for the overall performance, can be wrapped inside `Commands` or `Strategies`

and it can be substituted with more efficient implementations without implying changes in the upper layer.
Moreover, while freedom is left to define the abstract interface of a new polymorphic `BaseMethod` or `Strategy`, the interface of a `Command` is frozen, it defines only three actions, as shown in C.L. 2.26.

```
  // --- Command.hh --- //
class Command {
public:
  // constructor, destructor
  virtual void setup()   = 0; // setup private data
  virtual void unsetup()= 0; // unsetup private data
  virtual void execute()= 0; // execute the action
```

Code Listing 2.26: Command class definition

In COOLFluiD, managing the collaboration between different numerical methods is eased by the fact that `Commands` can create their own local or distributed data and share them with other `Commands` defined within other `Methods`, by making use of the `DataStorage` facility, whose details are presented in section 2.2.1.
Moreover, Perspective objects, as described in 2.3.1, can be used within the MCS pattern, at the `Strategy` level, as part of the `ConcreteMethodData`, in order to bind a numerical algorithm to the physics polymorphically.
This collaboration between MCS and Perspective patterns is an effective solution that decouples physics and numerics as much as possible. It makes possible, for instance, to employ a certain space method to discretize the equations related to different physical models, but also to apply different space methods, requiring completely different data-structures to discretize the same equations.
Interchangeability of `Methods`, `Commands`, `Strategies` can be facilitated and maximized by making them *self-registering* and *self-configurable* objects [19] [78, 80]. These last two concepts will be explained in Sec. 2.5. A deeper analysis of the MCS pattern and more implementation details are provided in [131, 132]. Among the possible `BaseMethods` that can be identified in a CFD simulation, and among those we have actually implemented in COOLFluiD, we consider now a few examples in order to show the concrete applicability of the MCS pattern.

2.4.1.1 MCS Example: SpaceMethod

In C.L. 2.27, we define the interface of a `SpaceMethod`, that takes care of the spatial discretization of the given set of PDE, according to a specified numerical scheme and on a chosen mesh.

```
// --- SpaceMethod.hh --- //
class SpaceMethod : public Method {
  public:
  SpaceMethod(string name);       // constructor
  virtual ~SpaceMethod();         // virtual destructor
  virtual void setMethod() = 0;   // setup the method
  virtual void unsetMethod() = 0; // unsetup the method

  // initialize the solution
  virtual void initializeSolution(CFbool isRestart) = 0;

  // compute the space part of residual/jacobian
  virtual void computeSpaceResidual(CFreal factor=1.0) = 0;

  // compute the time dependent part of residual/jacobian
  virtual void computeTimeResidual(CFreal factor=1.0) = 0;

  virtual void applyBC() = 0; // apply boundary conditions
};
```

Code Listing 2.27: SpaceMethod class definition

As shown in the sample code above, `SpaceMethod` inherits from a non instantiable `Method` object that provides some configuration functionalities meant to be reused by all its children, namely all the possible `BaseMethods` in Fig. 2.7.
Figure 2.8 shows a simplified class diagram of a concrete space method `ASpaceMethod` module. Specific `Commands`, `ASpaceMethodCom`, are associated to actions like *setup* and *unsetup* (creation and destruction of data needed by the employed scheme), application of *boundary conditions*, computation of the residual (plus jacobian contributions to the system matrix, in case of implicit schemes) coming from the space and time equation terms, as shown in C.L. 2.28.
All self-registering polymorphic objects, including `Commands`, are aggregated by `SelfRegistPtrs`, i.e. smart pointers with intrusive reference counting that keep ownership on them. In order to take full profit of the self-registration and self-configuration facility which will be described in Sec. 2.5, all `ConcreteMethods`, including `ASpaceMethod`, hold the `Command` names (second entry in the *std::pair* tuples), which are used as keys for the polymor-

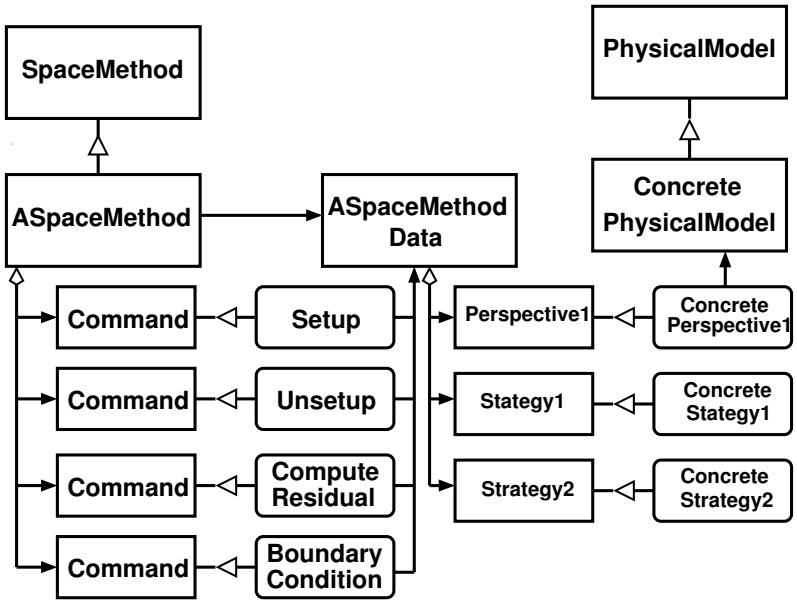

Figure 2.8: OMT diagram of the Method-Command-Strategy pattern applied to a concrete SpaceMethod, called ASpaceMethod.

phic creation and configuration of the `Commands` themselves. Let's consider the constructor of `ASpaceMethod`:
All the `Command` names are set to a default that can be overridden by the user in the COOLFluiD input file: these will cause the object requested by the user to be instantiated, if available. As clarifying example, let's analyze the following fragment of an input file:

```
SpaceMethod = ASpaceMethod
ASpaceMethod.SetupCom = SecondOrderSetup
ASpaceMethod.UnSetupCom = SecondOrderUnSetup
ASpaceMethod.ComputeRHS = JacobRHS
```

This asks to create the `SpaceMethod` corresponding to the name `ASpaceMethod` and to select *setup* and *unsetup* `Commands` specific for a second order scheme, which might require the allocation of different or additional data than a first order one. Likewise, the `Command` with name `JacobRHS` will be used to compute the residual and jacobian contributions instead of the default one, called `StdComputeRHS`.
All `AComs` are parameterized with a policy class [10], `ASpaceMethodData`,

```
// --- ASpaceMethod.hh --- //
class ASpaceMethod : public SpaceMethod {
  public:
  typedef SelfRegistPtr<Command<ASpaceMethodData> > ACom;
  // constructor, destructor, overridden virtual functions

  private:
  // data to share between ACom commands
  std::auto_ptr<ASpaceMethodData> m_data;

  std::pair<ACom,string> m_setup;   // setup Command
  std::pair<ACom,string> m_unsetup; // unsetup Command

  // Commands computing the residual/jacobian
  std::pair<ACom,string> m_computeSpaceRHS; // space part
  std::pair<ACom,string> m_computeTimeRHS;  // time part

  // Commands that initialize the solution in the domain
  std::vector<ACom> m_inits;
  std::vector<string> m_initsStr; // init Commands names

  // Commands that computes the boundary conditions (bc)
  std::vector<ACom> m_bcs;
  std::vector<string> m_bcsStr; // bc Commands names
};
```

Code Listing 2.28: ASpaceMethod class definition

```
// --- ASpaceMethod.cxx --- //
ASpaceMethod::ASpaceMethod(string name) :
SpaceMethod(name),
m_data(new ASpaceMethodData())
{
  m_setup.second = "StdSetup"; // default name
  addConfigOption("SetupCom","Setup",&m_setup.second);

  m_unsetup.second = "StdUnSetup"; // default name
  addConfigOption("UnSetupCom","UnSetup",&m_unSetup.second);

  m_computeSpaceRHS.second = "StdComputeRHS"; // default name
  addConfigOption("ComputeRHS", "Compute space residual",
  &m_computeSpaceRHS.second);
  ...
}
```

Code Listing 2.29: ASpaceMethod constructor

which groups together all the Strategy objects needed by the `Commands` to fulfill their job: `Strategy1`, `Strategy2`, etc. and some Perspectives, `Perspective1`, `Perspective2`, etc. (see 2.3.1), that provide the binding to the physics.

```
// --- ASpaceMethodData.hh --- //
class ASpaceMethodData : public ConfigObject {
  public:
  //constructor, destructor, configuration functions

  // accessor/mutators to Strategies
  SafePtr<Strategy1> getStrategy1() const;
  SafePtr<Strategy2> getStrategy2() const;

  // accessor/mutators to Perspectives
  SafePtr<Perspective1> getPerspective1() const;
  SafePtr<Perspective2> getPerspective2() const;

  // other accessors/mutators ...

  private:

  SelfRegistPtr<Strategy1> m_strategy1;
  string m_strategy1;
  ...

  SelfRegistPtr<Perspective1> m_perspective1;
  string m_perspective1;

  // ... the same for all the other objects
};
```

Code Listing 2.30: ASpaceMethodData class definition

A more detailed example showing the application of the pattern for implementing a FV method is given in Sec. 5.2.8. Other subclasses of SpaceMethod, such as `Finite Element` or `Residual Distribution`, are implemented in a similar way.

2.4.1.2 Collaboration between two Methods: ConvergenceMethod and LinearSystemSolver.

We consider now an example of interaction between two different numerical methods. Figure 2.9 shows the collaboration between two abstract methods: `ConvergenceMethod`, responsible of the iterative procedure, and `LinearSystemSolver`. In this case, `BackwardEuler`, an implicit conver-

gence method, delegates polymorphically the solution of the resulting linear system to `PetscLSS`, which interfaces the `PETSc` library [73].

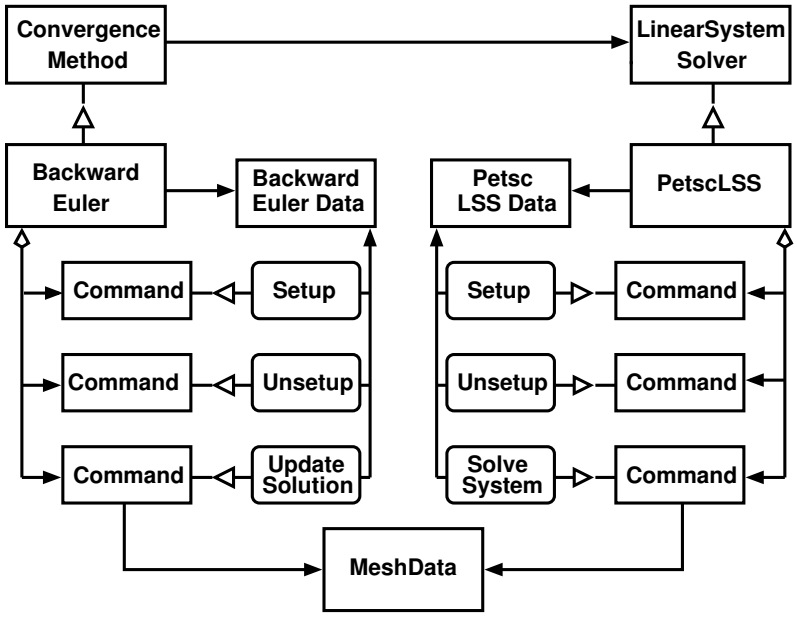

Figure 2.9: MCS pattern applied to two interacting modules: a Backward Euler convergence method and a Petsc linear system solver.

`BackwardEuler` makes use of `Commands` for the setup, unsetup and solution update, while `PetscLSS` let `Commands` implement the set up, unsetup and solution of the linear system.
The class definition of `ConvergenceMethod` is the following:
In the code above, `MultiMethodHandle` is a lightweight proxy object that hides the knowledge of multiplicity: it controls the access to one or more underlying Methods with the same polymorphic type and dispatches specified actions on all of them sequentially, similarly to what a *std::for_each* function would do [149]. The purpose of accessing `Method` objects through `MultiMethodHandles` is to offer transparent support for weakly coupled simulations, where, in the same process, two or more different linear systems are assembled by one or more `SpaceMethods` and must be solved one after the other.
`SpaceMethod` (SM) and `LinearSystemSolver` (LSS) are the collaborator Methods: a concrete `ConvergenceMethod` uses them polymorphically via `MultiMethodHandles`, without knowledge of their concrete type or their ac-

```
// --- ConvergenceMethod.hh --- //
class ConvergenceMethod : public Method {
public:
  //constructor, virtual destructor, configuration methods

  virtual void takeStep() = 0;    // Take one timestep
  virtual void setMethod() = 0;   // Sets up private data
  virtual void unsetMethod() = 0; // Clears up private data

  // Sets collaborator methods (SpaceMethod, LinearSystemSolver)
  void setCollaborator(MultiMethodHandle<SpaceMethod> spaceMtd);
  void setCollaborator(MultiMethodHandle<LinearSystemSolver>
      lss);
protected:
  // Synchronizes the states and computes the norm of the
      residual
  void synchAndComputeRes(CFbool computeResidual);

protected: //data
  // Space Method used to compute spatial discretization
  MultiMethodHandle<SpaceMethod> m_spaceMtd;

  // LinearSystemSolver used to solve the linear system (if any)
  MultiMethodHandle<LinearSystemSolver> m_lss;

  // handle to the global storage of states
  DataHandle<GLOBAL,CFreal> m_statedata;

  // handle to the global storage of nodes
  DataHandle<GLOBAL,CFreal> m_nodedata;
};
```

Code Listing 2.31: ConvergenceMethod class definition

tual number.
As a result, concrete collaborator `Methods` are completely interchangeable with other ones with the same polymorphic type. When running in parallel, in the function `synchAndComputeRes`() the global storage of state vectors and nodal coordinates are accessed via `DataHandle` and asked to synchronize the underlying parallel array, as shown in C.L 2.32.
In a parallel simulation, the norm of the residual is calculated with a collective operation, between the beginning and the end of the synchronization process, in order to overlap communication and computation and maximize the efficiency of the operation. This is one of the only few COOLFluiD code fragments that makes direct use of the global `DataHandles` and needs to conditionally apply a different treatment in parallel or in serial. We consider

```
// --- ConvergenceMethod.cxx --- //
void ConvergenceMethod::synchAndComputeRes(CFbool
    computeResidual)
{
  // after each update phase states and nodes
  // have to be synchronized
  if (isParallel()) {
    m_statedata->beginSync();
    m_nodedata->beginSync();
  }
  // computation of the residual
  if (computeResidual) {...}

  if (isParallel()) {
    m_statedata->endSync();
    m_nodedata->endSync();
  }
}
```

Code Listing 2.32: ConvergenceMethod synchronization

now a backward Euler time stepper, implemented as a subclass `BwdEuler` of the `ConvergenceMethod`:

```
// --- BwdEuler.hh --- //
class BwdEuler : public ConvergenceMethod {
public:
  //constructor, virtual destructor, configuration methods
  typedef SelfRegistPtr<Command<BwdEulerData> > BwdEulerCom;

  void takeStep();    // Take one timestep
  void setMethod();   // Sets up private data
  void unsetMethod(); // Clears up private data
private:
  //data shared by BwdEuler Commands
  std::auto_ptr<BwdEulerData> m_data;
  std::pair<BwdEulerCom,string> m_setup;     // set up
  std::pair<BwdEulerCom,string> m_unSetup;   // unsetup
  std::pair<BwdEulerCom,string> m_updateSol; // update solution
};
```

Code Listing 2.33: BwdEuler class definition

A different `Command` and its name are associated to each one of the pure virtual functions declared by the parent `ConvergenceMethod`. We report here after the implementation of `takeStep`(), where the polymorphic usage of the collaborators (SM and LSS) and of the solution updating `Command` is

shown.

```
// --- BwdEuler.cxx --- //
void BwdEuler::takeStep()
{
  // compute residual and jacobian contributions for
  // the spatial and time dependent term of the equations
  m_spaceMtd.apply(mem_fun(&SpaceMethod::computeSpaceResidual));
  m_spaceMtd.apply(mem_fun(&SpaceMethod::computeTimeResidual));

  // solve the resulting linear system
  m_lss.apply(mem_fun(&LinearSystemSolver::solveSys));

  m_updateSol.first->execute(); // update the solution

  // synchronize nodes and states, compute the intermediate
      residual
  ConvergenceMethod::synchAndComputeRes(true);
}
```

Code Listing 2.34: BwdEuler takeStep() function

The code is readable, concise, independent from the actual type and number of SM or LSS, extremely flexible, since each part of the algorithm (`Commands` or `Methods`) can be replaced at run-time without any performance overhead. In fact, the frequency of virtual calls in question is exceptionally low and, therefore, does not have an impact on the run-time speed.
The class definitions of the parent LSS and of its subclass `PetscLSS` are shown in C.L. 2.35 and 2.36.
`PetscLSS` delegates tasks to specific `Commands`,`PetscLSSCom`, sharing some data (`PetscLSSData`), such as references to the (parallel) Petsc matrix and the (parallel) Petsc vectors involved in the solution of the linear system.
Only the `PetscLSSCom` can make direct use of PETSc [73] objects like `PC` or `KSP`, which are aggregated by `PetscLSSData`. The knowledge of a specific LSS, `PetscLSS`, in this case, is not assumed anywhere in the numerical modules, thanks to the use of abstractions such as `LSSMatrix`, `BlockAccumulator` and `LSSIdxMapping`. `LSSMatrix` is the parent system matrix from which `PetscMatrix` derives. `BlockAccumulator` bundles blocks of values to be inserted in the matrix. `LSSIdxMapping` stores a mapping from the local numbering to an optimal LSS-dependent global one.
As represented in Fig. 2.9, all the involved `Commands` have acquaintance of `MeshData`, which provides access to `DataStorage` and `DataHandles`, and can therefore use and modify bulk data, qualified by name and type, as explained in 2.2.1. In other words, while, on the one hand, each `ConcreteMethodData`,

```
// --- LinearSystemSolver.hh --- //
class LinearSystemSolver : public Method {
public:
  //constructor, virtual destructor, configuration methods

  virtual void solveSys() = 0;    // solve the linear system
  virtual void setMethod() = 0;   // setup private data
  virtual void unsetMethod() = 0; // clear up private data

  // create a block accumulator with ad-hoc internal storage
  virtual BlockAccumulator* createBlockAccumulator
  (int nbRows, int nbCols, int subBlockSize) const = 0;

  // accessor/mutator for local to global LSS index mapping
  SafePtr<LSSIdxMapping> getLocalToGlobalMapping()
  {return &_localToGlobal;}

  // get the system matrix
  virtual SafePtr<LSSMatrix> getMatrix() const = 0;
private:
  // idx mapping from local to LSS global
  LSSIdxMapping m_localToGlobal;
};
```

Code Listing 2.35: LinearSystemSolver class definition

such as `PetscLSSData` and `BackwardEulerData`, allows intra-Method sharing of ConcreteMethod-dependent data among Commands, on the other hand, `MeshData` is the vehicle for inter-Method data exchange.
The sample code in C.L. 2.38 should help clarifying the last statement in the case of two `Commands`, namely `UpdateSol` in `BackwardEuler` and `SolveSys` in `PetscLSS`.

Data stored in `MeshData` are allowed to cross the Method scope and can be used in `Commands` belonging to different `Methods`. The keys for the data exchange are the storage name and type, that must both match in all the method `Commands` that need the same data.
In parallel simulations, nothing changes, since the `Commands` implementing numerical algorithms work only with `LOCAL` data, as they would do in a serial run. Functions that demands access to the `GLOBAL` storage and perform some parallel action, like the above mentioned `synchAndComputeRes`(), are exceptional.
Furthermore, the example of the collaboration between `ConvergenceMethod` and LSS demonstrates the suitability of the MCS pattern to interface existing libraries, PETSc in this case, without exposing any detail of their actual

```
// --- PetscLSS.hh --- //
class PetscLSS : public LinearSystemSolver {
public:
  //constructor, virtual destructor, configuration methods
  typedef SelfRegistPtr<Command<PetscLSSData> > PetscLSSCom;

  void setMethod();   // setup private data
  void unsetMethod(); // clears up private data
  void solveSys();    // solve the linear system

  // create a block accumulator with ad-hoc internal storage
  BlockAccumulator* createBlockAccumulator() const;

  // get the system matrix
  SafePtr<LSSMatrix> getMatrix() const
  {return &m_data->getMatrix();}

private:
  ///The data to share between PetscLSSCom Commands
  std::auto_ptr<PetscLSSData> m_data;
  std::pair<PetscLSSCom,string> m_setup;    // setup Command
  std::pair<PetscLSSCom,string> m_unSetup;  // unsetup Command
  std::pair<PetscLSSCom,string> m_solveSys; // solver Command
};
```

Code Listing 2.36: PetscLSS class definition

```
// --- PetscLSSData.hh --- //
class PetscLSSData {
public:
  //constructor, destructor, configuration methods

  PetscVector& getSolVec() {return m_xVec;} //Petsc solution
      array
  PetscVector& getRhsVec() {return m_bVec;} //Petsc rhs array
  PetscMatrix& getMatrix() {return m_aMat;} //Petsc matrix
  PC& getPreconditioner() {return m_pc;} //Petsc preconditioner
  KSP& getKSP() {return m_ksp;} //Petsc Krylov solver

  // accessors to various Petsc parameters, private data, etc.
};
```

Code Listing 2.37: PetscLSSData class definition

implementation to their clients.
The *Trilinos* package [72] has also been successfully integrated in COOLFluiD by means of the MCS pattern. While the class definitions of the corresponding concrete linear system solver `Method` and the related `Commands` are basi-

```
// --- UpdateSol.cxx --- //
void UpdateSol::execute()
{
  // get the local state vectors and rhs from MeshData
  DataHandle<LOCAL,State*> states = MeshData::getLocalData()->
    getData<State*>("states");

  DataHandle<LOCAL,CFreal> rhs = MeshData::getLocalData()->
    getData<CFreal>("rhs");

  // compute the solution update ...
}
```

Code Listing 2.38: UpdateSol execute() function

```
// --- SolveSys.cxx --- //
void SolveSys::execute()
{
  // get the local state vectors and rhs from MeshData
  DataHandle<LOCAL,State*> states = MethData::getLocalData()->
    getData<State*>("states");

  DataHandle<LOCAL,CFreal> rhs = MethData::getLocalData()->
    getData<CFreal>("rhs");

  // get the method data shared by all PetscLSS Commands
  PetscMatrix& mat = getDataPtr().getMatrix();
  PetscVector& rhsVec = getDataPtr().getRhsVector();
  PetscVector& solVec = getDataPtr().getSolVector();
  KSP& ksp = getDataPtr().getKSP();

  // perform final assembly and ask PETSc to solve the system
}
```

Code Listing 2.39: SolveSys execute() function

cally similar to the Petsc's ones, the interface of `TrilinosLSSData` is defined in C.L. 2.40.

In summary, the application of the MCS pattern helps to encapsulate both the single numerical algorithms and their collaborations with other classes of schemes. This contributes to enforce the multi-component-oriented character of the COOLFluiD framework, where each component can be replaced by another one with the same polymorphic type, without affecting the client code.

```
// --- TrilinosLSSData.hh --- //
class TrilinosLSSData {
public:
  // map for the individual IDs of the unknowns
  Epetra_Map* getEpetraMap() {return m_map;}

  TrilinosVector* getSolVec() {return &m_xVec;} //solution
      vector
  TrilinosVector* getRhsVec() {return &m_bVec;} //rhs vector
  TrilinosMatrix* getMatrix() {return &m_aMat;} //system matrix
  AztecOO* getKSP() {return &_ksp;} // Aztec Krylov solver

  // various options and parameters to control convergence
  // and tune the solver to satisfy the user needs ...
 };
```

Code Listing 2.40: TrilinosLSSData class definition

2.5 Dynamical plug-ins

In COOLFluiD, each concrete numerical method or physical model is enclosed in a separate *plug-in* library or module. This plug-in policy, which provides our platform with significant modularity and extensibility, relies heavily on two complementary techniques, namely *self-registration* and *self-configuration* of objects, whose basic principles re explained in this section.

2.5.1 Self-registering objects.

The self registration of objects, pioneered by [19] in a C++ context, automatizes the creation of polymorphic objects so that

1. the details of the concrete types are not exposed to the interface of the core code;
2. all possible objects are created using the same interface;
3. the core implementation should be unaffected by future unpredictable extensions of the code.

As a result, both implementation and compilation dependencies are considerably reduced, allowing developers to enclose each new functionality as a separate component, compile it separately and loading it in dynamically on demand into the platform, without even needing a partial recompilation. Our implementation of this technique, which is of great help in easing the

integrability of new components in COOLFluiD, has been inspired by [39], but we have evolved it into a more sophisticated and flexible approach, only partially explained in [74, 78] which is described here after.
A singleton [48] `Factory` class for a generic polymorphic object type (here represented by the template parameter `OBJ`) is defined in C.L. 2.41.

```
// --- Factory.hh --- //

template <class OBJ>
class Factory {
public:
  // get a reference to the Factory itself
  static Factory<OBJ>& getInstance()
  {static Environment::Factory<BASE> obj; return obj;}

  // register a provider
  void regist(Environment::Provider<OBJ>* provider)
  {m_map[provider->getName()] = provider;}

  // get the provider corresponding to the given key name
  SafePtr<typename OBJ::PROVIDER> getProvider(string pName)
  {return dynamic_cast<typename OBJ::PROVIDER*>
    (m_map.find(pName)->second);}

private:
  std::map<string, Provider<OBJ>*> m_map; // providers database
};
```

Code Listing 2.41: Definition of Factory class

Factory encapsulates an associative container, a std::map, which stores pairs key-value, where the key is a literal string and the value is a pointer to a `Provider` for objects of polymorphic type `OBJ`. Each `Provider` of a polymorphic object of dynamic type `OBJ` registers its name and itself in the corresponding `Factory` class at construction time, as shown in C.L. 2.42.
Another intermediate abstract class, called `ConcreteProvider`, needs to be defined: it derives from Provider and defines a virtual `create` function. In fact, several template specializations [160] of `ConcreteProvider` are provided, one for each foreseen number of parameters (0,1,2) accepted by the `create` function. Two of those class specializations are shown in C.L. 2.43. The type `SelfRegistPtr` appearing in C.L. 2.43 is just a proxy pointer class for handling self-registering objects in safety.
The implementation for the `create` functions defined by `ConcreteProvider` is given by `ObjectProvider` in C.L. 2.44, which also needs to be partially specialized for accomplishing the task.

```
// --- Provider.hh --- //
template <class OBJ>
class Provider {
public:

  // constructor: each Provider registers itself in Factory
  Provider(const string& name) : m_name(name)
  {Factory<OBJ>::getInstance().regist(this);}

  virtual ~Provider() {} // virtual destructor
  string getName() {return m_name;} // returns the name

private:
  string m_name; // provider name
};
```

Code Listing 2.42: Definition of Provider class

```
// --- ConcreteProvider.hh --- //
template <class OBJ, int NBARG = 0>
class ConcreteProvider : public Provider<OBJ> {
public:
 // create function accepting 0 arguments
 virtual SelfRegistPtr<OBJ> create() = 0;
};

template <class OBJ>
class ConcreteProvider<OBJ,1> : public Provider<OBJ> {
public:
  // definition of the object type and argument type
  typedef OBJ BASE_TYPE;
  typedef typename OBJ::ARG1 BASE_ARG1;

  // create function accepting 1 argument of type BASE_ARG1
  virtual SelfRegistPtr<OBJ> create(BASE_ARG1 arg1) = 0;
};
```

Code Listing 2.43: ConcreteProvider class

In particular, `ObjectProvider` is parameterized with three template arguments: the concrete type `CONCRETE`, its corresponding polymorphic type `OBJ` and the number of arguments for the creational function.
We now consider an example to show how the whole machinery works in practice. We define an abstract class `Shape` whose constructor accepts one string parameter. In order to be able to apply the self-registration of all its possible derived classes, the typedefs `PROVIDER` and `ARG1` must be explicitly

```
// --- ObjectProvider.hh --- //

template <class CONCRETE, class OBJ, int NBARGS = 0>
class ObjectProvider : public OBJ::PROVIDER {
public:
  // implementation of the create function with 0 arguments
  SelfRegistPtr<OBJ> create()
  {return SelfRegistPtr<OBJ>(new CONCRETE(), this);}
};

template <class CONCRETE, class OBJ>
class ObjectProvider<CONCRETE, OBJ, 1> : public OBJ::PROVIDER {
public:
  // implementation of the create function with 1 arguments
  SelfRegistPtr<OBJ> create(typename OBJ::ARG1 arg)
  {return SelfRegistPtr<OBJ>(new CONCRETE(arg), this);}
};
```

Code Listing 2.44: ObjectProvider class

provided with public access in the `Shape` class definition, as shown in C.L. 2.45.

```
// --- Shape.hh --- //

class Shape {
public:
  typedef ConcreteProvider<Shape, 1> PROVIDER;
  typedef string& ARG1;

  Shape(string& name);
  virtual double getVolume() = 0;
};
```

Code Listing 2.45: Shape class

At this point, whatever shape object (`Cube`, `Circle`, `Polyhedron`, etc.) deriving from the parent `Shape` can be self-registered by instantiating a global variable with type equal to the corresponding `ObjectProvider`, somewhere in the code (e.g. in the implementation file):
The registration of all available `ObjectProviders` occurs before entering the `main()` function of the code, since `Factories` live on the static memory and `ObjectProviders` are instantiated as global variables.
In our example, the string `"Cube"` (or `"Circle"`), accepted by the corresponding provider constructor, can then be used as a key to ask the singleton `Factory` to query its database and create the corresponding polymorphic

```
// --- Cube.cxx --- //
ObjectProvider<Shape, Cube, 1> cubeProvider("Cube");

// --- Circle.cxx --- //
ObjectProvider<Shape, Circle, 1> circleProvider("Circle");
```

Code Listing 2.46: Instatiation of a ObjectProvider for a Cube in the implementation file

object whenever needed.

```

string keyName = shapeName(); //can be "Cube" or "Circle" etc.

SelfRegistr<Shape> shape = Factory<Shape>::getInstance().
                getProvider(keyName)->create(keyName);
```

Code Listing 2.47: Creation of a polymorphic shape object

This example shows that whatever the actual shape, the way to create it remains unique and the piece of code in C.L. 2.47 doesn't need to be changed anymore even if new shapes are developed and integrated in the platform.
In COOLFluiD, all polymorphic objects are created with this technique, once their corresponding names (e.g. `keyName` in the example in C.L. 2.47) are read from the CFcase input file.

2.5.2 Self-configurable objects.

In COOLFluiD, objects can be self-configurable [78, 80, 131], i.e. they can create and set their own data. In order to make an object self configurable, some steps have to be followed. First, the object must derive from the abstract parent class `ConfigObject`, which implements the configuration algorithm [131].
Second, a static member function `defineConfigOptions()` must be defined and implemented. The implementation consists in declaring the type, name and description for each one of the options, as in the illustrative example in C.L. 2.48 for a `TimeStepper` object:
Once declared, the options must be linked to the actual data to configure and this is done inside the constructor, as in the example in C.L. 2.49.
In particular, the strings "CFL" and "LSS" in our example are the configuration keys and they are used to map the values of the private data `m_cfl` and `m_lssName` with the ones read from the configuration case file (CFcase).

```
// --- TimeStepper.cxx --- //
void TimeStepper::defineConfigOptions(OptionList& op)
{
  op.addConfigOption<double>
                ("CFL", "CFL number");
  op.addConfigOption<string>
                ("LSS", "Linear System Solver name");
}
```

Code Listing 2.48: Example of implementation of `addConfigOption()` function for a self-configurable TimeStepper

```
// --- TimeStepper.cxx --- //
TimeStepper::TimeStepper() : ConfigObject()
{
  addConfigOptionsTo(this);

  m_cfl = 1.0; // default CFL value
  setParameter("CFL", &m_cfl);

  m_lssName = "PETSC"; // default LSS name
  setParameter("LSS", &m_lssName);
}
```

Code Listing 2.49: Example of configuration directives in the constructor of a self-configurable TimeStepper

In order to configure our `TimeStepper` object, the end-user should write something like

```
TimeStepper = BwdEuler  # name of the concrete TimeStepper
BwdEuler.CFL = 3.5      # user-defined CFL number
BwdEuler.LSS = TRILINOS # user-defined linear system solver
```

in the CFcase file. `BwdEuler` is the self-configuration value for `TimeStepper` which is the configuration key for the homonymous polymorphic object. `BwdEuler` is also the self-registration key (see Sec. 2.5.1) for the concrete time stepper class that will then be instantiated and will configure itself with the specified CFL number and linear system solver name.
In COOLFluiD, Methods, Commands, Strategies, PhysicalModels etc. are all self-configurable objects and this, together with the self-registration technique, gives the end-user full power on their settings. The actual implementation of the described technique, due to [131], allows users to input whatever

kind of data (including analytical functions) from file, environmental variables or command line options, but also to define interactive parameters or to substitute on-the-fly polymorphic objects during the simulation.