The COOLFluiD Framework: Design Solutions for High Performance Object Oriented Scientific Computing Software

Andrea Lani[1], Tiago Quintino[1,2], Dries Kimpe[2,3], Herman Deconinck[1], Stefan Vandewalle[2], and Stefaan Poedts[3]

[1] Von Karman Institute, Aerospace Dept., Chaussee de Waterloo 72, B-1640 Sint-Genesius-Rode, Belgium
[2] Catholic University Leuven, Computer Science Dept., Celestijnenlaan 200A, B-3001 Leuven, Belgium
[3] Catholic University Leuven, Center for Plasma-Astrophysics, Celestijnenlaan 200B, B-3001 Leuven, Belgium

Abstract. The numerical simulation of complex physical phenomena is a challenging endeavor. Software packages developed for such purpose should combine high performance and extreme flexibility, in order to allow an easy integration of new algorithms, models and functionalities, without penalizing run-time efficiency. COOLFluiD is an object-oriented framework for multi-physics simulations using multiple numerical methods on unstructured grids, aiming at satisfying these needs. To this end, specific design patterns and advanced techniques, combining static and dynamic polymorphism, have been employed to attain modularity and efficiency. Some of the main design and implementation solutions adopted in COOLFluiD are presented in this paper, in particular the Perspective and the Method-Command Patterns, used to implement respectively the physical models and the numerical modules.

1 The COOLFluiD Architecture

COOLFluiD (**C**omputational **O**bject-**O**riented **L**ibrary for **Flui**d **D**ynamics) is a multi-physics and multi-methods platform that combines flexibility and high performance for the simulation of complex fluid dynamical phenomena on unstructured grids. The package is implemented in C++, which, during the last decade, has shown a great potential for scientific applications, offering significant support to develop both flexible and efficient code: *Cogito*, *ELEMD* ([Arge97]), *MOUSE*, *Deal* ([OONum]) are only few examples of available C++ platforms.

An overview of the COOLFluiD framework is sketched in Fig. 1. It consists of a kernel, where *Simulation*, the simulation manager object, and *MeshData*, the basic data-structure object are implemented. Also the abstract interfaces for all the polymorphic objects are defined in the kernel, in particular the ones for the physics description (*PhysicalModel*) and for the numerical

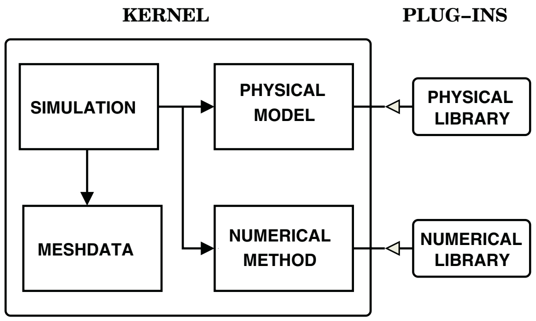

Fig. 1. Simplified overview of the COOLFluiD framework

methods (e.g., *MeshCreator*, *SpaceMethod*, *ConvergenceMethod*, *LinearSystemSolver*). Each concrete numerical method or physical model is enclosed in a separate *plug-in* library. This *plug-in* policy, which provides COOLFluiD with significant modularity and extensibility, relies heavily on two complementary techniques, namely *self-registration* and *self-configuration* of objects, whose basic principles are explained below.

1.1 Self Registration of Objects

The *self registration of objects*, pioneered by [Bev98] in a C++ context, automatizes the creation of polymorphic objects and reduces implementation and compilation dependencies. This is of great help in easing the integrability of new components in the framework, since they can be compiled as external *plug-ins* and loaded dynamically, on demand, into the main core application. A generic polymorphic concrete object (*ConcreteObj*) can be registered by simply instantiating the corresponding *ObjectProvider* in the implementation file:

```
ObjectProvider<BaseObj, ConcreteObj> myProvider("objName");
```

and it can be created by calling the corresponding *Factory*:

```
BaseObj* objPtr = Factory<BaseObj>::getProvider("objName")->create();
```

1.2 Self Configuration of Objects

In COOLFluiD, objects can be self-configurable, meaning that they can create and set their own data. The template configuration function was inspired by the *Yagol* library [Yagol]. An object is made self configurable, by deriving it from a parent class *ConfigObject* and by adding a call to

```
addConfigOption("OptionKey", &configData);
```

in its constructor for each configurable data member *configData*. In particular, *OptionKey* is the configuration key string, used to map the value of *configData*. We consider, for instance, what would appear in a configuration file for the *RK* (Runge-Kutta) time stepper object:

```
ConvergenceMethod = RK
RK.coeff = 0.28 0.61 1.0
```

RK is the self-configuration value for the *ConvergenceMethod*, which is the configuration key for the homonymous polymorphic object. *RK* is also the self-registration key for the Runge-Kutta class, that will be then instantiated and will configure itself with the three given coefficients, whose configuration key is *coeff*.

1.3 Parallel Data-Structure

COOLFluiD uses a parallel layer designed to minimize impact on both users and software developers by exporting a high level platform independent interface. Parallelization is fully transparent and high performance is assured by techniques like parallel IO and remote memory access, when supported by the underlying platform. The intrinsically parallel data-structure is encapsulated in a *Facade* object [Gamma95], *MeshData*, whose main component is *DataStorage*. The latter offers a simple interface to create and handle generic typed arrays of data to be shared among different numerical *Methods* and *Commands*, allowing to treat uniformly both local and distributed data. When running in parallel, all MPI calls are encapsulated in an underlying dynamically growing parallel array, hidden to the clients of *DataStorage*. The following examples show how to create and get local (global) data:

```
getDataStorage()->createData<StorageType>("storageName",
                                          storageSize);
DataHandle<LOCAL, StorageType> myStorage =
  getDataStorage()->getData<StorageType>("storageName");
```

2 Physical Model: Perspective Pattern

The framework is designed to apply different numerical methods for the solution of systems of *Partial Differential Equations*, which typically appear in the form:

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \mathbf{F} = \mathbf{S} \qquad (1)$$

where $\mathbf{U}$ (unknown variables), $\mathbf{F}$ (convective and viscous fluxes), $\mathbf{S}$ (source term) depend on the chosen physical model. Such a physical model can be seen as a composition of entities, e.g., coefficients, quantities and thermodynamic properties. Note that one can look at the same physics through different formulations of the equations, involving the use of different sets of variables, transformations, or other adaptations tailored, e.g., towards a particular numerical method.

This logical picture can be translated into an implementation offering multiple views or *perspectives* for the same physics, according to the specific needs. The first advantage is that this design approach is able to break a hypothetical heavy and hard-to-define interface for the physical model object into a limited

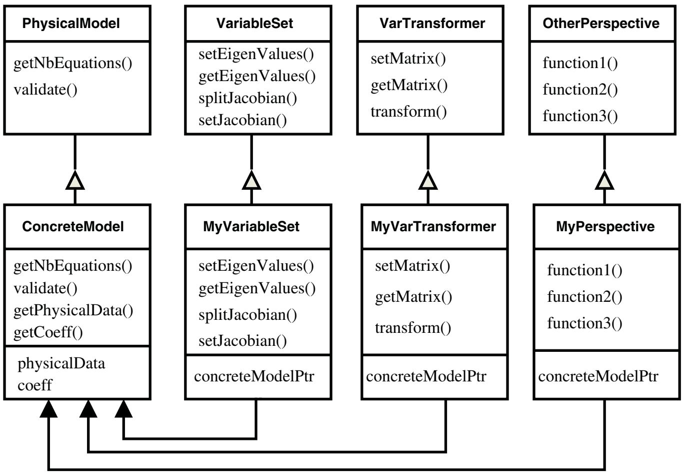

Fig. 2. Perspective pattern applied to a Physical Model module

granularity of independent abstractions. Flexibility and maintainability are positively affected too. If new base *Perspectives* objects are needed in order to provide new points of view for the same physics, they can easily be attached without needing to modify the existing code.

The OMT (Object Modeling Technique) class diagram of the pattern is shown in Fig. 2. It reflects the composition-based *Adapter* described in [Gamma95], but it has a single shared *Adaptee* object (*ConcretePhysicalModel*), and multiple abstract *Targets* (called *Perspectives* here), each one with a number of derived classes (*Adapters*). The *Perspective* pattern is conceptually opposite to the *View Handler* presented in [Busch00], since the former helps to tackle a situation in which the different views (and their interfaces) are not all foreseeable a priori, meaning that it would be impossible to define both an handler object and a single abstract view interface, as required by the latter.

Data and functionalities that are typical of a certain physics, but invariant to all its possible *Perspectives*, should be implemented in the actual *ConcretePhysicalModel* object. The base *PhysicalModel* defines a very general abstract interface. The concrete one implements the virtual methods of the parent class and defines another interface to which the *ConcretePerspective* objects (and only those) are statically bound. As a result, a client makes use of the physical model through an abstract layer, enlargeable if required, given by a number of perspective objects (*VariableSets*, *VariableTransformers*, etc.) while all their collaborations with the concrete physics are completely hidden. As all the other polymorphic objects in COOLFluiD, also *PhysicalModels* and *Perspectives* are self-registrable and, if needed, self-configurable.

3 Method-Command Pattern

Every numerical module is implemented following a common design structure, to which we will refer as the *Method-Command* pattern. It consists of a concrete *Method* object delegating tasks to a number of *Commands* that share a tuple of multiple receivers. A configurable *BaseMethod* object (e.g., *SpaceMethod*, *ConvergenceMethod*, *MeshCreator*) defines the abstract interface for a specific type of Method (Fig. 3). Each *ConcreteMethod* implements the virtual functions of the corresponding parent *BaseMethod* by encapsulating requests for specific actions (setup, unsetup, compute something . . .) in ad-hoc *Commands*. *ConcreteMethod* functions can act as *Template Methods* [Gamma95] where the hooks are not virtual functions but polymorphic commands. Each *Command* behaves therefore as a *Strategy* object (see [Gamma95]) in performing a specific task, which can be accomplished in several different ways. All the *Commands* share some common data enclosed in *ConcreteMethodData*, a configurable tuple typically aggregating the multiple receivers. The latter are polymorphic *Strategy* or *Perspective* objects, providing, e.g., the dynamic binding to the physics.

The flexibility yielded by this structural pattern is considerable and does not affect performance, since the fast-path code is wrapped inside the concrete *Commands* or their receivers. The *Method-Command* pattern can be viewed as a sophisticated variant of the *Whole-Part* pattern described in [Busch00], or as a three-layer *Strategy*, where *Methods*, *Commands* and command receivers are completely interchangeable, self registering and self configurable.

Some applications of the pattern within the COOLFluiD framework will now be presented, in order to show the high reusability provided by this approach, but also its suitability to deal with complex numerical problems.

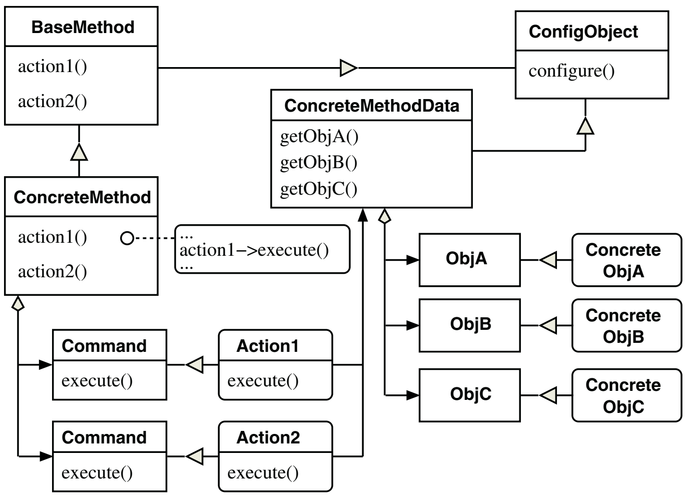

Fig. 3. OMT diagram of a Method-Command pattern

3.1 Space Method

A *SpaceMethod* takes care of the spatial discretization of the given set of partial differential equations, according to a specified numerical scheme on a given mesh. The abstract interface of a *SpaceMethod* is as follows:

```
class SpaceMethod : public ConfigObject {
public:
  // constructor, destructor, accessors, mutators ...
  virtual setup() = 0;    // setup data
  virtual unsetup() = 0;  // unsetup data
  virtual void computeRHS() = 0; // compute residual and jacobian terms
  virtual void applyBC() = 0; // apply boundary conditions
};
```

A possible concrete class is the so-called cell centered *Finite Volume* (FV) method:

```
class CellCenterFVM : public SpaceMethod {
public:
   typedef Command<CellCenterFVMData> FVMCom;
   // overridden parent virtual functions ...
private:
  SharedPtr<CellCenterFVMData> _data; // shared data
  std::auto_ptr<FVMCom> _setup;   // setup command
  std::auto_ptr<FVMCom> _unSetup; // unsetup command
  std::auto_ptr<FVMCom> _compRHS; // compute residual command
  std::vector<FVMCom*>  _bcs;     // boundary conditions
};
```

As an example, we present the implementation of the method *applyBC()* in *CellCenterFVM*, which shows how all actions are nicely encapsulated:

```
void CellCenterFVM::applyBC() {
  for_each(_bcs.begin(), _bcs.end(), mem_fun(&FVMCom::execute));
}
```

As shown in Fig. 4, *CellCenterFVMData* holds pointers to polymorphic Strategies like *FluxSplitter*, the flux splitting scheme and *PolyRec*, the polynomial reconstructor; *VarSet*, the Perspective encapsulating physical model traits related to specific sets of variables, etc.

The implementation of other concrete *SpaceMethods*, like *Residual Distribution* (RD) or *Finite Element* (FE), follows an identical *Method-Command* pattern.

3.2 ConvergenceMethod and LinearSystemSolver

Figure 5 shows the collaboration between two abstract methods: *Convergence Method*, responsible of the iterative procedure, and *LinearSystemSolver*. In this case, an implicit convergence method, *BackwardEuler*, delegates polymorphically the solution of the resulting linear system to *PetscLSS*, which interfaces

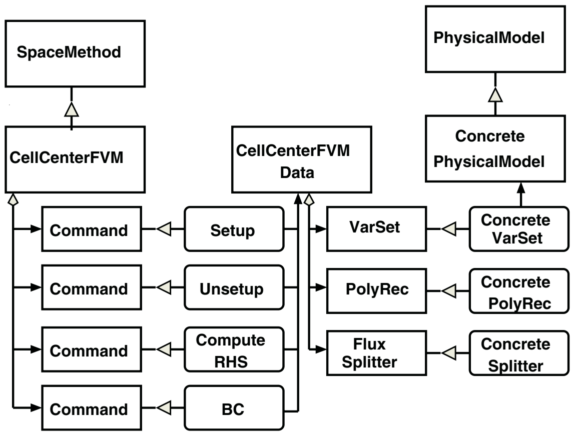

Fig. 4. Finite Volume module

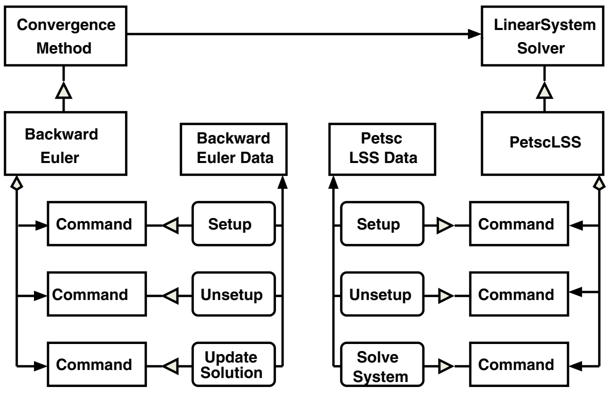

Fig. 5. Backward Euler and Petsc Linear System Solver modules

the *PETSc* library ([Petsc]). *BackwardEuler* makes use of commands for the setup, unsetup and solution update. Also *PetscLSS* delegates tasks to specific commands 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.

4 Conclusions

The flexible and reusable design solutions presented in this paper have allowed an easy integration of several components in COOLFluiD: explicit (Runge-Kutta) and implicit (Newton, Crank-Nicholson) time stepping, different spatial discretizations (FV, RD, Space-Time RD, FE), different physical models (Euler, compressible Navier-Stokes, Magneto Hydro Dynamics), a parallel flexible data-

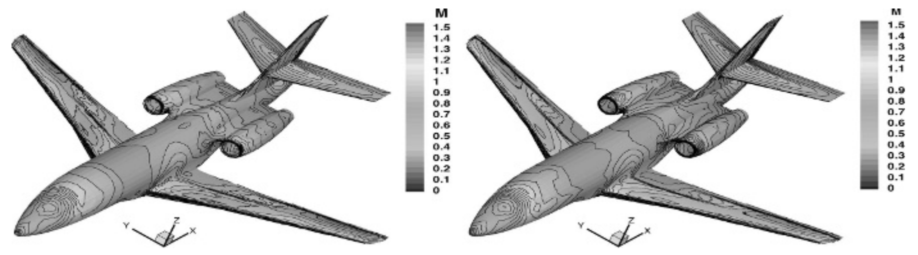

Fig. 6. Mach contours and isolines for an Euler simulation with blended LDA/N scheme (left) and 2nd order Roe scheme with Barth limiter (right) on a Falcon airplane (mesh courtesly provided by Dassault Aviation)

structure supporting the use of hybrid meshes, etc. The implementation of many other functionalities is underway: Aero-Thermo-Chemical models, incompressible Multi-Phase flows, error estimation, mesh movement and adaptation.

Fig. 6 shows the result of a simulation of the Euler gas dynamics equations over a Falcon airplane geometry, flying at Mach 0.85, with two different high order spatial discretizations, namely cell vertex RD and cell centered FV.

Acknowledgments

T. Quintino acknowledges the financial support of Fundaćao Ciencia e Tecnologia.

References

[Bev98] Beveridge, J.: Self-Registering Objects in C++. Dr. Dobbś Journal, 8/1998.

[Gamma95] Gamma, E., Helm, R. Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.

[Busch00] Buschmann, F., Meunier, R. Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software architecture. A System of Patterns. Wiley, 2000.

[Arge97] Arge, E., Bruaset, A. M. , Langtangen, H. P. eds.: Modern Software Tools for Scientific Computing, Birkhäuser, 1997.

[OONum] Open Systems Laboratory: The Object-Oriented Numerics Page, http://www.oonumerics.org/oon/, 2005.

[Yagol] Pace, J.: Another Getopt Library, http://yagol.sourceforge.net, 2003.

[Petsc] Argonne National Laboratory: PETSc. Portable, Extensible Toolkit for Scientific Computation, http://www-unix.mcs.anl.gov/petsc, 2004.