# Object-Oriented Programming Applied to the Finite Element Method
## Part I. General Concepts

**Jacques Besson * — Ronald Foerch ****

** Ecole des Mines de Paris, Centre des Matériaux
CNRS URA 866, F-91003 Evry cedex*

*** North West Numerics Inc., 444 N.E. Ravenna Blvd
Suite 301-A, Seattle, WA 98115, USA*

ABSTRACT. *This paper examines the application of object-oriented programming techniques to the finite element method. First a tool library is briefly presented: it includes mathematical objects such as vector, matrix and tensor, as well as generic types such as array, list and encapsulated pointer. Design patterns are then presented. They allow the defining of reusable implementation strategies which help to obtain a flexible extensible code. Finally the paper demonstrates the use of the different patterns in the case of objects describing finite elements and object representing material behaviors.*

RÉSUMÉ. *Cet article examine l'application des techniques de programmation orientée objet à la méthode des éléments finis. En premier lieu, une bibliothèque d'utilitaires est brièvement présentée. Elle comprend des objets mathématiques tels les vecteurs, les matrices et les tenseurs, ainsi que des types génériques tels que les tableaux, les listes ou les pointeurs encapsulés. Des schémas de conception sont ensuite présentés. Ils permettent de définir des stratégies d'implantation réutilisables qui permettent d'obtenir un code flexible et extensible. Finalement, l'article montre l'utilisation des différents schémas dans le cas des objets de type « éléments finis » et des objets de type « comportement ».*

KEY WORDS: *object oriented languages, finite element method, programming patterns, constitutive equations, C++.*
MOTS-CLÉS : *langages orientés objet, méthode des éléments finis, schémas de programmation, équations de comportement, C++.*

## 1. Introduction

The need for highly structured, extensible and reusable simulation tools for engineers and scientists has been an important issue for a long time. Early in the 60s, large finite element (FE) codes were developed using FORTRAN (IV, then 77). Although this language allows for a structured programming, it became soon obvious that it was lacking some very useful characteristics such as the to handle of handling sets of data as typed structures or easy dynamic memory allocation.

In the 80s, solutions have been proposed to solve these difficulties: Specific languages dedicated to the Finite Element Method (FEM) were introduced to ease programming. These languages were based on FORTRAN and allowed both the use of structured data sets and dynamic memory allocation. Indeed, the C language was already offering these possibilities. However, the programming effort needed to rewrite thousands of existing FORTRAN lines was thought prohibitive compared to the cost of developing pre–processors allowing to translate these specific languages into native FORTRAN. These pre–processors were similar to "C–front" pre–processors which were first developed to translate C++ into C before compilation. However, these specific languages were limited to procedural programming.

At the end of the 80s, Object Oriented Programming (OOP) techniques were first applied to the FEM using languages such as C++, Smalltalk or Eiffel. Objects consist in both data and methods allowing the manipulation of these data. They also implement inheritance, polymorphism and abstraction [RUM 91]. A programming language which does not allow these functionalities, should not be considered as an object–oriented language. Therefore, the difference between C and C++ is much more important than the difference between FORTRAN and C. OOP was first applied to windowing systems, text processors or data bases [GAM 94, BOO 97]. It was shown efficient in solving problems encountered using procedural programming such as easily supporting multiple window systems. However, the use of OOP to scientific programming such as the FEM, still remains very limited. The main limitation is indeed the cost of (re)designing new softwares. Obviously, a mere translation from FORTRAN to C++ would not achieve the desired goals.

The present research work about the application of OOP to the FEM, was carried out using the FE software Zébulon developed by the Centre des Matériaux since 1982. The redesign of the code started in 1992. This paper first presents a definition of objects with respect to structures. Base objects such as vectors, files, arrays, etc., are then described. The second part of the paper deals with generic programming patterns and demonstrates their use in the case of elements and material behaviors.

## 2. Structures and Objects

It is first necessary to define what objects are and, in particular, to present the difference between objects and structures. A structure is essentially a set of data. This data can either be types defined by the language (`int`, `double`, `char`, `char*`, etc.) or other structures. Procedures are used to handle these structures. This technique is used in FE codes such as CASTEM2000 [ VER 88] and SIC [ BRE 92].

Objects contain both data sets and member functions used to handle their data. With these functionalities only, an object would be equivalent to the set {data + procedures}. The new features proposed by object–oriented languages (OOL) are [ STR 88, RUM 91, BOO 97]:

— The ability to **encapsulate** data and member functions (`public`, `protected`, `private` et `const` declarations).

— The possibility to use **inheritance** to define object hierarchies. An object can have many children which inherit its data and member functions. It can also have one or several parents (single or multiple inheritance).

— The possibility of defininf **abstract** objects using virtual member functions. This is one of the most important characteristics of OOLs. Abstract objects allow the writing of generic algorithms and the easy extension the existing code. The object then have a **polymorphic** behavior.

These characteristics are mandatory in order to have a true object–oriented language. Some other features can greatly ease programming such as:

— The ability to overload member functions and operators such as `=`, `+`, `+=`, `<`, etc.

— Object and function templates (generic types) can also be included. These are, for instance, generic arrays. Templates greatly ease and improve the use of objects while requesting few lines of code.

Java, for the sake of simplicity, does not allow overloading and templates. Operator overloading appears to be mandatory in the case of an OOL applied to scientific computations.

In the following, objects will be designated using uppercase letters, and instances using lowercase letters. For instance `OBJ` will represent an object:

| | |
|---|---|
| `OBJ obj1,obj2;` | declaration of two instances `obj1` and `obj2` of `OBJ`. |
| `obj1=obj2` | affectation of an instance. |

C++ or C++–like notations will be used.

## 3. Choosing a Programming Language

The usefulness of OOP for scientific computations has nowadays been recognized. However actual programming still faces the problem of choosing a language. It is also obvious that the programming language can influence the code organization and even the used algorithms.

FORTRAN 77, C and FORTRAN 90 are clearly not OOLs. It is however possible, by storing function pointers in structures, to create structures that simulate the behavior of objects. This has the advantage of easily reusing the existing code; on the other hand it is undoubtedly a very constraining technique as the programmer has to perform a part of the task that is devoted to the compiler. Ada has also been applied to the FEM [ LUC 92, LUC 94]; however the language does not easily allow obtaining inheritance and abstraction. The numerical efficiency of the resulting software remains poor. In addition, the availability of Ada compilers remains limited. One can also notice the use of LISP [ MIL 88], which remains undoubtedly limited.

The use of Smalltalk has been carefully examined by Zimmermann *et al.* [ ZIM 92, DUB 92b, DUB 92a]. This language is simple and emphasized the key concept of OOP: encapsulation, inheritance, abstraction [ DUB 93]. It remains, however, very slow as it is interpreted and not compiled.

C++ seems to be, nowadays, the language allowing both numerical efficiency and the use of OO techniques. Efficiency relies on low-level programming similar to C. Polymorphism is the main cause that can lower performances as it requires run time linking. A C++ code can therefore use high-level abstraction while optimizing some critical steps. This is clearly the case of the FEM where an important fraction of the CPU time can be spent in solving linear systems such as $[K]\{U\} = \{F\}$. In addition, C++ is widely used and available on numerous platforms. The standards are not fully defined which can sometimes cause porting problems. As far as numerical efficiency is concerned, it is clear that C++ compilers do not optimize the code as much as FORTRAN compilers which has been developed for more that 40 years [ HAN 94, STR 94]. Availability, wide use and efficiency dictated the choice of C++ for the development of Zébulon.

Java has been recently introduced [ LAF 97, COM 96] as a platform independent language. It has a C++-like syntax. It implements garbage collecting which does not exist in C++ as it is time consuming [ STR 94]. Java supports neither multiple inheritance nor templates nor operator overloading. This can be cumbersome in the case of scientific programming. Pointers are missing; this somehow eases programming but limits code optimization.

## 4. Programming Pattern

Programming pattern are programming methods which allow solving, in a somehow standardized way, problems which are often encountered [ GAM 94]. For clarity, patterns are usually named (e.g., *composite*, *visitor*, etc.). Patterns proposed in literature, were initially proposed and applied to text processors, window managers or drawing tools [ GAM 94, BOO 97]. Patterns are indeed not limited to OOP; dealing with memory allocation in FORTRAN often obeys a pattern.

Problems encountered during the development of Zébulon were, in most cases, related to the organization of the different objects and to their interactions, so that it could be possible to obtain a high factorization of tasks while keeping the ability of extending an existing hierarchy without affecting the existing code.

As it was often noticed [ RUM 91, BOO 97], OOP follows (undoubtedly more than procedural programming) a cyclic design so that object hierarchies or the organization must be regularly modified. A typical cycle period is between 6 months to one years; it tends to increase as the project matures (or gets out of control). A whole object organization often collapses when new functionalities are added to the code. This cannot be avoided in the case of a research code which does not have well predefined objectives on a long time period. The following programming patterns address these problems.

The code also uses some "basic tools" such as vector, matrix, arrays, strings, etc. They are briefly presented in Appendix A. In the following, the code examples will use these objects.

### 4.1. *Object Hierarchies and Factorization*

Figure 1 shows three types of object hierarchies:

Single inheritance. This is the simplest organization (tree–like) where an object has only one parent. The base class `A` is usually abstract. Zébulon uses this construction in most cases.

Multiple inheritance. This is an object hierarchy where each object can have one or more parents. There are many base classes (`A` and `B`) which are often abstract. The result is usually a complex construction which is difficult to maintain (see Figure 1). This construction is not often used in Zébulon. It is used when the different base classes fulfil separate tasks. In most cases, one of the classes, is a utility class. For instance, material behaviors derive from both the base class `BEHAVIOR` and a class encapsulating a numerical integration method such as `RUNGE_KUTTA` or `THETA_METHOD`.

Virtual base class. This is probably the most complex construction for object hierarchies. Base classes `A` et `B` both derive from class `V` which is virtual.

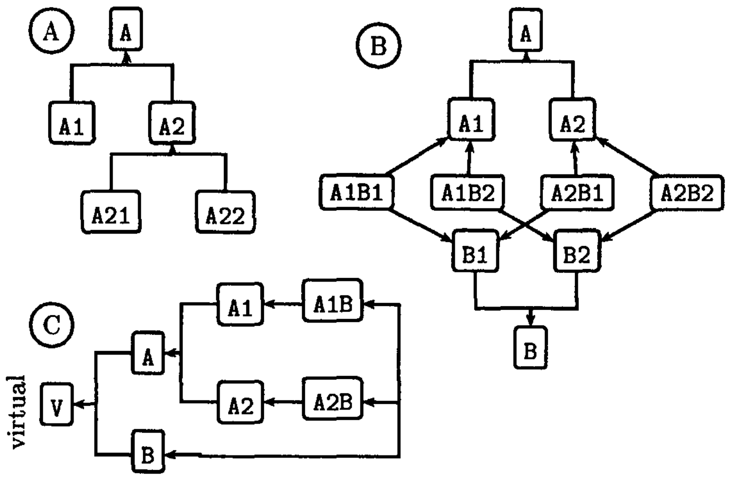


**Figure 1.** *Three organizations of object hierarchies: (A) single inheritance, (B) multiple inheritance, (C) multiple inheritance with virtual base class (class* `V`*)*

(see [STR 86] Sections 6.5.1 and 6.5.3). The result is highly complex programming which (after some unfruitful attempts) was not used in Zébulon. We believe that it can only be used in the case of well designed class hierarchies which will not be modified (for instance the base class `ios` of C++).

Note that multiple inheritance is not supported by all OOL: Java prohibits it. Some papers dealing with the application of OOP to the FEM (e.g., [KON 95]) propose complex object hierarchies based on multiple inheritance which leads to objects such as:

```
ELEMENT_AXI_8NODE_ELASTIC
```

This indeed leads to an unnecessary large number of classes. One should never forget the difference between inheritance and data member. It is clear that a car **has** tires and does not **derive** from tire. In the previous case, `ELEMENT` should not derive from `ELASTIC` but should have a behavior as a data member (even if it is purely elastic). Note, indeed, that each Gauss point can have a different behavior.

```
class CAR :
   public TIRE
{...};
```

NO

```
class CAR {
   ARRAY<TIRE> tires;
};
```

YES

In an object hierarchy, the base class plays a particular role as it defines the **user interface** for all derived objects. The interface consists in all the

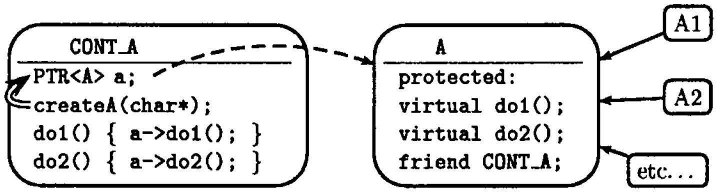


**Figure 2.** *Encapsulation of a hierarchy (base* A*) by a container class* CONT_A

member functions of the base class and in particular in the virtual functions which can be redefined by the different derived classes. It defines a grammar for the hierarchy user. In order to ease and secure the use of interface, the hierarchy can be encapsulated in a container class. In the example shown on Figure 2, the hierarchy derived from the base class A is encapsulated by the class CONT_A which is declared friend. The whole interface of A is protected so that is can only be accessed by CONT_A. Instances of CONT_A are initialized by the member function createA which creates an instance of type A using for instance a key word. This function often uses the *object factory* pattern (see below).

**Factorization** is a corollary of object organization using hierarchies. The role of factorization is to avoid, when adding a new object to the hierarchy, to duplicate existing code to implement the new object functionalities. The scenario, which is often encountered, is summarized in the following (see Figure 3). The initial hierarchy (base B) is, for instance, defined as follows:

```
class BASE {               class B1 :               class B2 :
   virtual f()=0;            public  BASE {           public  BASE {
};                             virtual f();             virtual f();
                           };                       };
```

A new object B3 is added; however its member function f() strongly duplicates the f() function defined in B1 (this usually results in a large "Copy&Paste"). Indeed, all "bugs" contained in B1::f() are consequently also infecting B3::f(), causing numerous maintenance problems. To avoid this problem, a new intermediate object BC is introduced in the hierarchy. f() is now implemented at this level; it uses new virtual functions (fbc) allowing the specification of the differences between B1 and B3:

```
class BC :                 class B1 :               class B3
  public  BASE {             public  BC {             public  BC {
   virtual f();               virtual fbc();          virtual fbc();
   virtual fbc()=0;        };                       };
};
```

BASE and B2 remain unchanged. The process is indeed somehow intrusive; in some cases it is necessary to modify the arguments of the interface functions

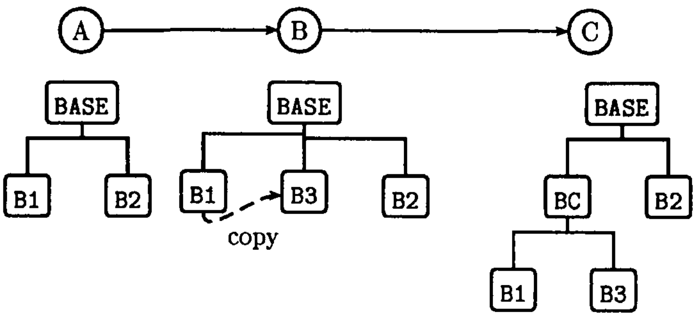


**Figure 3.** *An example of code factorization: (A) initial hierarchy, (B) hierarchy after adding the new object* B3, *(C) hierarchy after factorization of classes* B1 *and* B3 *using* BC

(i.e. f) and consequently lead modifications at several locations in the code. We, nethertheless, believe that this greatly helps in keeping a clean code and often leads to simplifications and clarifications.

## 4.2. *Creation of Objects (Object Factory)*

Patterns related to object creation are used to obtain an abstract object instantiation. It helps to make the system independent in the way the different objects are created. Some methods (creational patterns) are listed in [ GAM 94].

The problem encountered most during the development of Zébulon is schematically depicted on Figure 4. A new object (BEXP) has to be added to an existing hierarchy (base object BASE), but it should be used by the rest of the code without being explicitely named. These problem is indeed solved, as far as its use is concerned, thanks to the common user interface of the hierarchy declared in BASE using virtual member functions. The problem remains in the case of the creation of an instance of BEXP from a character string (read for instance in the input file). The most obvious solution consists in declaring a static member function belonging to the base class BASE which returns a pointer on an object of type BASE.

```
BASE* BASE::create_object(cont STRING& name) {
   if     (name=="b1")   return new B1();
   else if(name=="b2")   return new B2();
   else if(name=="bexp") return new BEXP();
   else                  return NULL;
}
```

This solution considerably limits the creation of librairies and the extensibility of the code. For instance, the base object BASE can represent the base for all elements used in the FEM. In order to use the previous creational

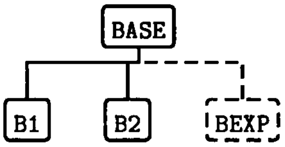


**Figure 4.** *Adding a new "experimental" class* BEXP *in an existing object hierarchy*

pattern, the implementation of the base class should include the definition of all derived classes: mechanical, thermal, post-processing, etc., elements. Which means that it would include almost all the class declarations of the code. This simple pattern should clearly be limited to very specific hierarchies.

One solution consists in centralizing object creation in one object: the Object Factory [ BES 97]. The method uses functions returning `void*` pointers. Their type (OBJECT_FACTORY_FUNC) is defined using a `typedef` declaration:

```
typedef void* (*OBJECT_FACTORY_FUNC)(void);
```

Such a function has then to be associated to a base class, a derived class and a key word. For instance, in the case of the new class BEXP:

```
void* OF_BASE_BEXP_bexp() { return new BEXP; }
```

A global instance of a utility class (OBJECT_FACTORY_PIECE) is then declared; the arguments of its creator are the name of the base class, the key word and the address of the function used to create the instances (i.e., OF_BASE_BEXP_bexp):

```
OBJECT_FACTORY_PIECE
    OFP_BASE_BEXP_bexp("BASE","bexp",OF_BASE_BEXP_bexp);
```

During the instantiation of OFP_BASE_BEXP_bexp, the different arguments of its creator are stored by the class OBJECT_FACTORY using its interface function `declare_object`:

```
 OBJECT_FACTORY_PIECE::OBJECT_FACTORY_PIECE(
       const STRING& base,
       const STRING& keyword,
       OBJECT_FACTORY_FUNC f)
{ OBJECT_FACTORY::Instance().declare_object(base,keyword,f); }
```

The member function `Instance()` enforces the fact that the OBJECT_FACTORY class has only one instance [ GAM 94]. The OBJECT_FACTORY class also has a member function (`make_object`) which is used to instantiate objects using the name of a base class and a key word. This function only retrieves the function pointer (type OBJECT_FACTORY_FUNC) associated to both names and calls the function. This allows theo centralization of the object creation. The return argument of `make_object` is a `void*` pointer which has to be explicitly cast:

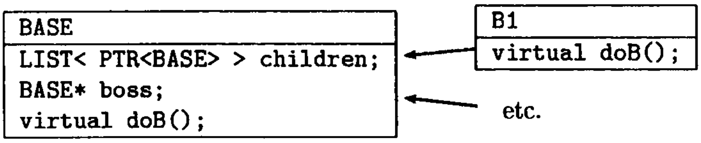
```
STRING key=...; // read in input file
BASE* b=(BASE*)OBJECT_FACTORY::Instance().make_object("BASE",key);
b->initialize(...);
```

where key is a key word which can be read in the input file. The whole process results in a "virtual constructor" of classes derived from BASE. Specific "macros" are defined to hide the declaration of functions of type OBJECT_FACTORY_FUNC and of object of type OBJECT_FACTORY_PIECE to make the pattern user-friendly. In addition, macros help to limit the risk of wrong implementation (such as misspelling key words).

It is important to notice that all objects created using the proposed *object factory* pattern must have a constructor taking no arguments. Instantiation must therefore be followed by an initialization. This can be seen as a limitation but turns out to be an advantage as initialization can then make use of virtual functions which cannot be called during construction. Separating instantiation from initialization is probably a good programming rule.

## 4.3. *Composite*

The *composite* pattern is frequently used. It allows the representation of hierarchies as tree structures. This methodology is easily applied to text processors or graphics applications [ GAM 94]. In an object hierarchy, the base class defines a set of virtual functions (doB) and manages the storage of object of the same type (children). The base class is therefore used both as a primitive and a container. The base class can also contain a pointer on the object it depends on (boss); this defines a fully recursive tree:



The implementation of the function doB in the base class generally consists in applying the same function to the stored objects. Derived classes call the method of the base class and execute their own instructions:

```
BASE::doB()
{ for(int i=0;i<children.size;i++) children[i]->doB(); }

B1::doB()
{ BASE::doB();
  // my own stuff
}
```

## 4.4. *Association*

It is often necessary, in order to define an object, to inherit properties from different objects. Multiple inheritance is the usual solution to this problem; however, as already mentioned, it can cause several difficulties. One of them is encountered (Figure 5), when "mixed" objects `A&B` are to be created from two (or many) existing object hierarchies (`A` and `B`). The solution, based on multiple inheritance, consists in explicitly implementing all possible associations: `A1B1...AnBm`. This is indeed very constraining and induces strong dependences between the different modules. Extensibility is therefore reduced.

A much more flexible solution relies a dynamic association where both hierarchies are cooperating. Two implementations are however possible. In the first case, one object controls the other one. The master class `A` contains an encapsulated pointer on the slave class `B` which contains a pointer on its master. Both base classes usually define virtual functions `doA()`, `doB()`). `A` can behave like `B` (and vice–versa) using the following method:

```
class A {                          class B {
   PTR<B> its_B;                      A* its_A;
   virtual doA();                     virtual doB();
   doB() { its_B->doB(); }            doA() { its_A->doA(); }
};                                 };
```

In the second case, it is not desirable to define a master object. A container object is then defined which holds instances of `A` and `B`. Using the same technique as in the previous case, the resulting object behaves as desired. The *object factory* pattern eases the initialization of the `AB` object.

```
                                AB::initialize(...)
class AB {                      { ...
   PTR<A> its_A;                   its_A=Create_object(A,keyA);
   PTR<B> its_B;                   its_A->initialize(...);
   doA() { its_A->doA(); }         its_B=Create_object(B,keyB);
   doB() { its_B->doB(); }         its_B->initialize(...);
   initialize(...);                ...
};                              }
```

`Create_object` is a macro which eases the use of `OBJECT_FACTORY`. `keyA` (resp. `keyB`) is a key word specifying the object deriving from `A` (resp. `B`) to be created.

Inside an *association*, the collaboration between associated objects is usually done using interface functions. In order to avoid numerous function calls and data copying, which may slow down the computation, data members can be

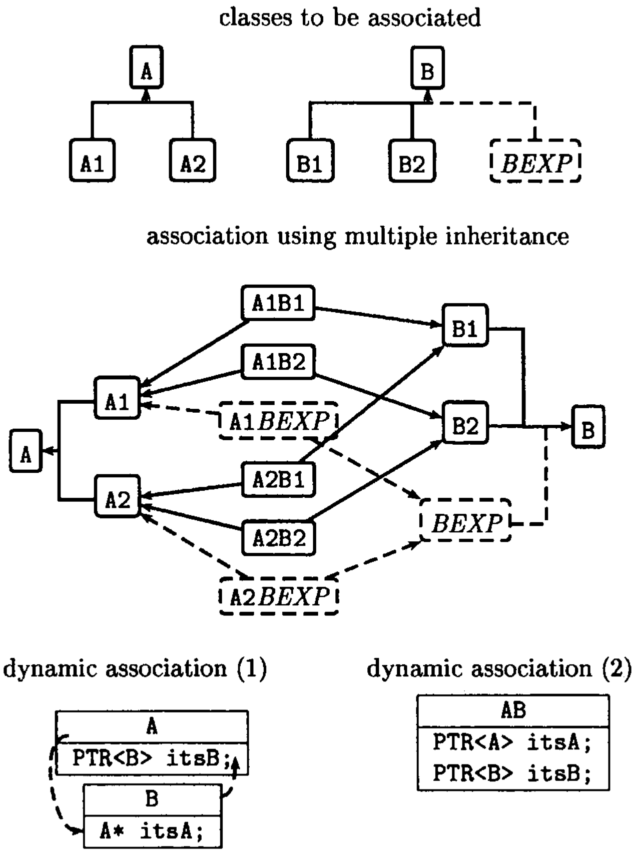


**Figure 5.** *Three ways of associating objects*

shared between the different objects.

```
class A {                class B {
  PTR<B> itsB;             A* itsA;
  DATAB* dataB;   →        DATAB  dataB;   = A&B
  DATAA  dataA;   ←        DATAA* dataA;
};                       };
```

In this example, the data member `A::dataB` (resp. `B::dataA`) points on `B::dataB` (resp. `A::dataA`). In this case, encapsulation is intentionally broken. The result is a more efficient code, which is usually more difficult to understand and maintain.

## 4.5. *Run time type identification. Transversal interrogation of an object hierarchy*

C++ allows, thanks to `dynamic_cast<>` and `typeid` functions, the identification of object types during execution. The introduction of these functionalities in the language was necessary as C++ is compiled and not interpreted [ STR 94]. Problems envisaged in this section are therefore not relevant in the case of Smalltalk. Unfortunately, few C++ compilers implements Run Time Type Identification (RTTI). It is however, easy to partially simulate this behavior with virtual functions using character strings to name the different objects.

The use of RTTI can usually be avoided (and should be) thanks to virtual functions. An object hierarchy is designed to perform a limited number of tasks. However, when the hierarchy is large, some of the derived objects can implement functionalities whose factorization, using an additional virtual function in the base class, would corrupt the hierarchy. For instance, in the case of the material behavior hierarchy, this could be the computation of the elastic deformation, the plastic dissipation, etc. In the case of finite element (base class `ELEMENT`, see Figure 6), it can be the calculation of the stored energy or the energy release rate [ PAR 74, DEL 85]. In the case of the calculation of Rice $J$ integral [ RIC 68, DEL 85], only few elements can perform this task. The calculation is done after convergence and is quick. It is therefore possible to use, without risk, time consuming methods. The calculation consists in a volume integral over an element set (`ELSET`$\rightarrow$`ARRAY<ELEMENT*>`) surrounding the crack tip. In this example, it is assumed that the standard "small deformation" element (`MCESD_STD`) can perform the task. The implementation could look like:

```
double J_INTEGRAL::compute(const ELSET& set)
{ double J=0.; MCESD_STD* elem;
  for(int i=0;i<set.size;i++) {
      elem = dynamic_cast<MCESD_STD*>(set[i]);
      if(!elem) ERROR("element is unable to compute J integral");
      J += elem->compute_J_integral();
  }
  return J;
}
```

This solution helps avoid perturbing the hierarchy while implementing a minor functionality (transversal interrogation). However, in the case where the computation of the $J$-integral has to be implemented for another element type the previous function has to be modified. Note that the factorization of the functionality by the introduction of a "mechanical element" allowing, at this level, the definition of a virtual function called `compute_J_integral`, would also fail in the case of elements implementing strong thermo-mechanical coupling. In addition, the proposed factorization could conflict with other needs.

A more flexible method, but slightly more difficult to implement, consist in defining a virtual function in the base object (`virtual void ask(MESSAGE&);`)

which allows the possibility to "ask" an object any question. This function can indeed be overloaded in order to be able to respond a particular request. The default implementation in the base class issues an error message. The implementation of the calculation of the $J$–integral, uses the following code:

```
double J_INTEGRAL::compute(const ELSET& set)
{ double J=0.;
  T_MESSAGE<double> Jask("J-integral");
  for(int i=0;i<set.size;i++) {
      set[i]->ask( Jask );
      J += Jask.x;
  }
  return J;
}
```

The MESSAGE object and the template class T_MESSAGE<T> are defined as follows:

```
class MESSAGE {
 public :
  STRING message;
  MESSAGE(const STRING& msg)
    { message=msg; }
};
```

```
template class <T> class T_MESSAGE
 : public MESSAGE {
 public :
  T x;
  T_MESSAGE(const STRING& msg) :
     MESSAGE(msg) {}
};
```

The member function MCESD_STD::ask is then implemented as follows:

```
void MCESD_STD::ask(MESSAGE& msg)
{ if(msg.message=="J-integral") {
     T_MESSAGE<double>* _msg =  dynamic_cast<T_MESSAGE<double>*>(&msg);
     if(!_msg) ERROR("wrong message type");
     compute_J_integral(*_msg);
  } else MCESD::ask(msg);
}
```

MCESD is the object, MCESD_STD immediately derives from. This method is safe as it uses dynamic_cast<> which allows the checking of the message type. The only possible error would be to give a wrong message name (here J-integral) or to misspell it. This error will be detected at run time as it will be impossible to interpret the message. This simulates the behavior of an interpreted language.

## 5. Examples of Object Hierarchies

In this section, a part of Zébulon is described in order to illustrate the previous programming patterns. Two parts of the code are described: the elements and the material behaviors. The *object factory* pattern has been widely used in the code, in particular in order to create association and composites.

The first papers dealing with the application of OOP to the FEM, immediately introduced objects representing nodes, elements, meshes, degrees of freedom (dofs) [FOR 90, MAC 92, DUB 92a, CAR 94, KON 95]. In some cases, the object description also included CAM/CAD concepts in order to create a graphic interface [BET 96, GAJ 96]. Zébulon uses similar concepts but the resulting objects slightly differ from their precursors.

Papers dealing with material behaviors are very few. In most cases, the material is assumed to be isotropic linear elastic so that the behavior is included in the element which has data members representing the Young's modulus and the Poisson's ratio. A treatment of the Von Mises plasticity with isotropic hardening is proposed by [MEN 93]. This formulation does not however clearly separate the behavior from the FEM. Consequently, the behavior object `J2Plasticity` handles Gauss points and the class `GaussPoint` checks if the yield surface is reached (function `GaussPoint::computeTrialStress`). The result is an interdependent structure which is likely not to be flexible and difficult to extend.

In all cases, published papers on OOP applied to the FEM, deal with prototypes whose aim is to demonstrate the feasibility of such a project, to propose some objects or to estimate computational efforts. However, difficulties tend to rapidly increase as the code grows and as new unplanned functionalities are added [BOO 97]. The principal danger is the duplication of existing code. While developing Zébulon, we have experienced the cycles described by Booch [BOO 97]. The code presently manages non-linear mechanical, thermal and diffusional problems. It includes a large number of constitutive equations. Besides solving FE problems, it deals with pre-and post-processing and optimization problems. It includes about 150,000 C++ lines.

The present description of the elements is the result of

## 5.1. *Elements*

numerous iterations. Comparison with earlier papers [AAZ 93, BES 97] shows the evolution. Two main lessons can be drawn from this experience: reorganizing the existing software is necessary if it avoids the treatment of special cases and code duplication; this reorganization is usually done quite easily if done every 6 months or year (in particular, if compared to a FORTRAN code).

Figure 6 depicts objects deriving from the base type ELEMENT. This hierarchy is an example of strong factorization. The main objects are the following:

**ELEMENT.** This object is responsible for the geometrical description of the element. It follows the ***association*** pattern according to the following diagram:

$$\boxed{\texttt{ELEMENT \& GEOMETRY \& INTEGRATION}}$$

ELEMENT is in charge of the element connection and contains pointers on its nodes and a pointer on the mesh to which it belongs. GEOMETRY deals

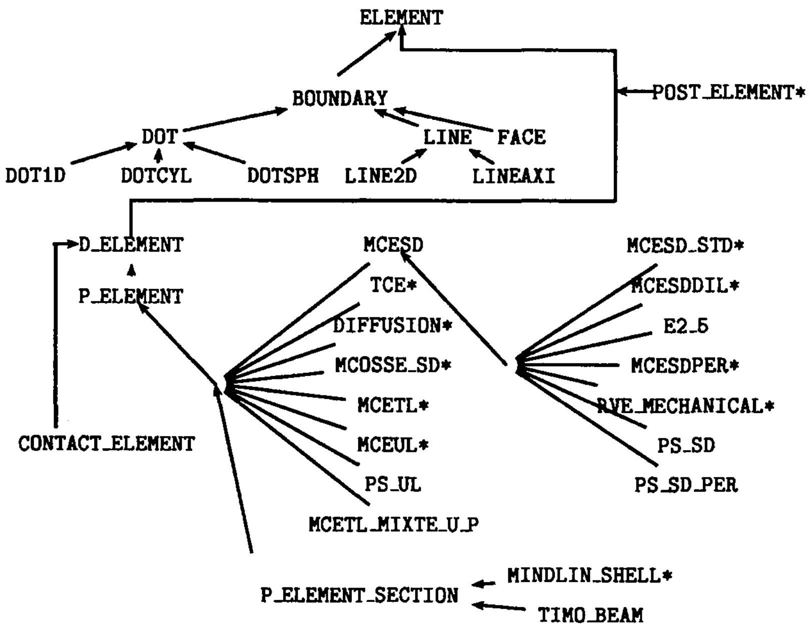
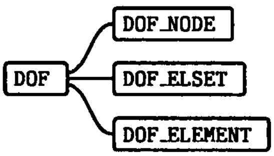


**Figure 6.** *Element hierarchy. * indicates that the tree has been truncated*

the actual geometrical description. It contains an interpolation function and is the base object of a hierarchy used to describe the different geometrical situations. It consists of a geometrical type: dot, 1D, spherical, cylindrical, 2D, axisymmetric or 3D, and of a space type: dot, line, surface, space. A 3D shell therefore corresponds to the (3D, surface) set. There is no need to use the *association* pattern to build all the different geometry/space couples, as they are well defined. INTEGRATION is in charge of the Gauss quadrature. It contains a set of reference Gauss points [ WAL 96]. It is important to note that none of the three parts of the association refers to a fixed number of nodes and fixed number of Gauss points. These are specified by the interpolation functions and the reference Gauss points. It is important to make the distinction between interpolation, quadrature and geometry. A plane continuous element and a 3D shell can share the same. Therefore the code does not implement such objects as "20 nodes/27 Gauss points" elements. It is dynamically created thanks to the association of several instances. The code can therefore be easily extended by adding new shape functions and reference Gauss points; in addition code duplication is avoided. In the association, ELEMENT is considered as the master object and therefore contains interface functions to access the other associated objects.

**BOUNDARY.** This object factorizes functionalities such as flux or pressure calculations, needing by boundary conditions. Each element is able to create a BOUNDARY object corresponding to one of its sides.

**D_ELEMENT.** This object handles degrees of freedom (dofs) associated to a variational principle. It therefore contains a new virtual interface function compute_R_K allowing to compute reactions associated to its dofs $\{R_e\}$ as well as the stiffness matrix of the element $[K_e]$. Dofs are represented by three objects:



dof associated to a node

dof associated to a group of elements (ELSET)

dof associated to an element

Dofs at nodes allows to discretize continuous fields such as displacements and temperatures. Dofs at ELSET are used, for instance, to represent generalized plane strain conditions. Dofs at elements are used, for instance, to deal with plane stress conditions [ BES 97].

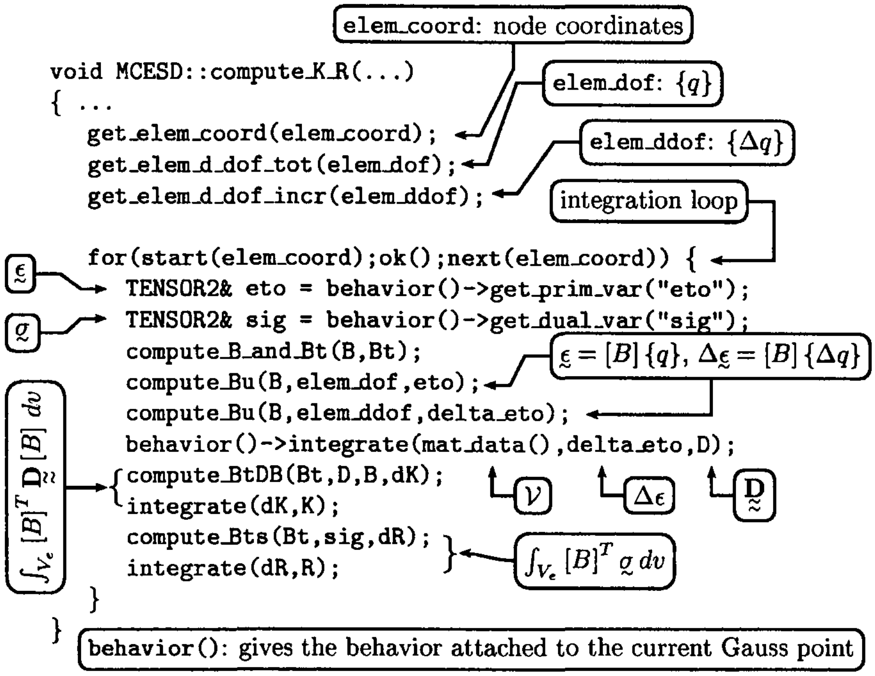
**P_ELEMENT.** This object introduces the behavior of the material constituting the element. It is important to make the distinction between D_ELEMENT and P_ELEMENT. For instance, contact elements deal with dofs but does not possess any behavior. In order to be able to deal with various situations, each Gauss point can have a different behavior. P_ELEMENT is also responsible of the storage on the variables describing the state of the material. It also manages the local material orientation.

**MCESD.** This object represents small deformation mechanical elements. Computation of the elementary stiffness matrix $[K_e]$ and of the internal forces $\{R_e\}$ can be summarized by the following steps:

```
time increment: $t \rightarrow t'$
loop over the Gauss points
    $\underset{\sim}{\varepsilon}^{t'} = [B]\{q\}^{t'}$
    behavior: $\underset{\sim}{\varepsilon}^{t'} \rightarrow \underset{\sim}{\sigma}^{t'}$
              $\mathcal{V}^{t} \rightarrow \mathcal{V}^{t'}$
    $[K_e] += [B]^T \underset{\approx}{\mathbf{D}} [B] \omega$
    $\{R_e\} += [B]^T \underset{\sim}{\sigma}^{t'} \omega$
```

| | |
|---|---|
| $\{q\}$ | vector of the element dofs |
| $\underset{\sim}{\varepsilon}$ | deformation tensor |
| $\underset{\sim}{\sigma}$ | stress tensor |
| $\mathcal{V}$ | state variables |
| $\underset{\approx}{\mathbf{D}}$ | tangent matrix |
| $\omega$ | "volume" attached to a Gauss point |

Figure 7 gives the source code needed to implement this element. The code uses factorized members functions such as get_elem_coord (returning the element nodes coordinates as a vector). The set start(.), ok(), next(.)



```
void MCESD::compute_K_R(...)
{ ...
   get_elem_coord(elem_coord);
   get_elem_d_dof_tot(elem_dof);
   get_elem_d_dof_incr(elem_ddof);

   for(start(elem_coord);ok();next(elem_coord)) {
     TENSOR2& eto = behavior()->get_prim_var("eto");
     TENSOR2& sig = behavior()->get_dual_var("sig");
     compute_B_and_Bt(B,Bt);
     compute_Bu(B,elem_dof,eto);
     compute_Bu(B,elem_ddof,delta_eto);
     behavior()->integrate(mat_data(),delta_eto,D);
     compute_BtDB(Bt,D,B,dK);
     integrate(dK,K);
     compute_Bts(Bt,sig,dR);
     integrate(dR,R);
   }
}
```

elem_coord: node coordinates
elem_dof: $\{q\}$
elem_ddof: $\{\Delta q\}$
integration loop
$\underset{\sim}{\epsilon}$
$\underset{\sim}{\sigma}$
$\int_{V_e} [B]^T \underset{\approx}{\mathbf{D}} [B]\, dv$
$\underset{\sim}{\epsilon} = [B]\{q\}$, $\Delta\underset{\sim}{\epsilon} = [B]\{\Delta q\}$
$V$
$\Delta\epsilon$
$\underset{\approx}{\mathbf{D}}$
$\int_{V_e} [B]^T \underset{\sim}{\sigma}\, dv$
behavior(): gives the behavior attached to the current Gauss point

**Figure 7.** *Source code implementing the small deformation element. Management of local material orientations has been omitted.*

manages the integration loop. The different `integrate` functions allow the integration of scalar, vectors and matrices over the element. This set, which is defined in the base class ELEMENT, frees the developer of the explicit management of volumes associated to each Gauss point. The function `get_elem_d_dof_tot` (resp. `get_elem_d_dof_incr`), defined in D_ELEMENT, allows the obtaining of get the values of the dofs at $t'$ (resp. the increment of the dofs over the time increment $t \to t'$) in the order of declaration. At this level, no specific assumption is made on the nature of the dofs. They will be specified in the different objects deriving from MCESD. In the case of the standard formulation (MCESD_STD) and in the case of selective integration [ HUG 80] (MCESDDIL) dofs are defined at nodes and represent displacements. Generalized plane strain elements (E2_6) and periodic elements [ BES 88] (MCESDPER) have dofs defined over an element set, representing the average deformation, and nodal dofs, representing the local perturbation (displacements). The RVE_MECHANICAL is an element whose geometry is a "dot" and whose dofs represent deformations. It allows, for instance, the testing of material constitutive equations [ LER 97].

Management of the plane stress conditions (PS_SD) is done without modifying the material constitutive equations as it is usually done. Dofs, representing the deformation in the plane stress direction, are associated with the element for each Gauss point in order to enforce the plane stress condition [ BES 97].

This solution allows the writing of the material behavior without any reference to the FEM. Using this formulation, any newly developed behavior can be immediately used in plane stress calculations. Finally, the plane stress periodic element (PS_SD_PER) uses the three different kinds of dofs: DOF_ELSET to represent the average deformation, DOF_NODE to represent the perturbation and DOF_ELEMENT to enforce the plane stress condition.

The $[B]$ matrix which related the dofs and the deformation tensor, is then defined according to the type of geometry. It is computed using the virtual function compute_B_and_Bt. Except for plane stress elements, it is then necessary to derive additional objects to specify the geometry: MCESD_STD_AXI, MCESD_STD_2D, MCESD_STD_3D,.... The virtual functions compute_Bu and compute_Bts are performing the products $[B]\{q\}$ and $[B]^T \underset{\sim}{\sigma}$.; their default implementation used the standard matrix×vector product. They can be overloaded it order to optimize this operation as the $[B]$ matrix contain a large number of null terms.

The evaluation of the material behavior is done using the member function BEHAVIOR::integrate which performs the evaluation of the stress tensor at the end of the increment $\underset{\sim}{\sigma}^{t'}$ as well as of the tangent stiffness matrix [ SIM 85]

$$\underset{\approx}{D} = \frac{\partial \Delta \underset{\sim}{\sigma}}{\partial \Delta \underset{\sim}{\varepsilon}}$$

Once again, it is important to note that the calculation of the element stiffness matrix and the internal reactions can be formulated independently of the space dimension and of the geometry. In addition, no specific assumption is made on the type of material behavior which is used, except that it is a "small deformation mechanical" behavior. An error message is issued if the behavior is not consistent with the element.

**Other elements.** Figure 6 indicates all physical elements (P_ELEMENT) presently implemented in Zébulon. All these elements obey the same rules as MCESD. In particular, the type of behavior should not be specified. They include thermal elements (TCE), diffusional elements (DIFFUSION), updated/total Lagrangian mechanical finite strain elements (MCEUL and MCETL). Plane stress conditions under finite strain, can be managed using the same method as for the small strain elements. MCETL_MIXTE_U_P represents total Lagrangian elements with mixed displacement–pressure formulation used to handle incompressible materials.

The MCOSSE_SD object represents the small deformation Cosserat elements [ DEB 91]. In this case, dofs represent displacements and micropolar rotations. They are associated with nodes. However, the variational principle is similar to the case of standard small deformation elements. This object can therefore be factorized using the MCESD object ( Figure 7). At the present time, this element and the associated material behaviors are still undergoing tests and evaluations [ FOR 97]; it will be factorized during the next development cycle.

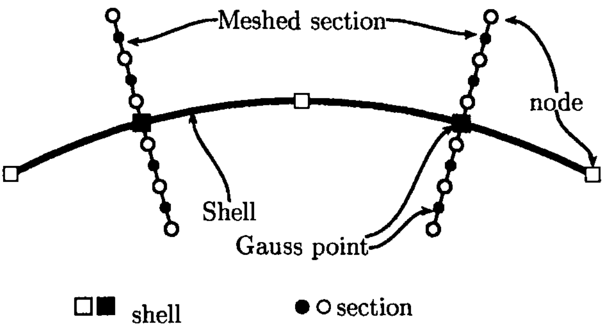


**Figure 8.** *Mesh of a 2D quadratic shell using reduced integration whose section consist in five linear element with one Gauss point*

MINDLIN_SHELL and TIMO_BEAM object represent Mindlin shells and Timoschenko beams [BAT 91]. These elements are very often used with elastic or perfectly plastic materials in order to be able to easily express the constitutive relation between forces/moments and displacements/rotations. It is, however, possible to reuse the proposed treatment of the plane stress conditions in order to be able to use any kind of behavior with shell and beam elements. In this case, cross-sections are meshed (P_ELEMENT_SECTION object); different nodes and Gauss points are then used to model the shell and its sections (Figure 8). For each Gauss point of the section, an extra dof (normal deformation) is added to obtain a null stress in the direction normal to the shell. An other solution consist in adding only one dof per section; in this case the normal deformation is assumed to be constant in the section. The average normal stress is null. The material behavior is then evaluated for each Gauss point of each section, using the same interface as in the case of the MCESD object (Figure 7).

Elements of type POST_ELEMENT are used for post-processing computations (such as some failure criterion). They have neither dofs nor behaviors, but specific data structures allowing the handling of results. As they inherit from the object describing the geometry of the element, they can easily perform volume integrals using the `start`, `ok`, `next` and `integrate` members functions already presented in the case of the MCESD element (Figure 7).

## 5.2. *Material Behaviors*

The various material behaviors were designed so that they do not refer to the FEM. This libraries can therefore be used in other softwares. Indeed, a generic interface for behaviors must be defined.

| **behavior type** | `primal` variable | `dual` variable |
|---|---|---|
| small deformation (mechanical) | $\underset{\sim}{\epsilon}$ | $\underset{\sim}{\sigma}$ |
| finite strain (mechanical) | $\underset{\sim}{\mathbf{F}}$ | $\underset{\sim}{\mathbf{S}}$ |
| Cosserat | $(\underset{\sim}{e}, \underset{\sim}{\kappa})$ | $(\underset{\sim}{\sigma}, \underset{\sim}{\mu})$ |
| thermal | $(T, \underline{\mathrm{grad}T})$ | $(H, \underline{q})$ |
| electrostatic | $\underline{\mathrm{grad}V}$ | $\underline{\mathbf{E}}$ |
| magnetostatic | $\mathbf{rot}\underline{\mathbf{A}}$ | $\underline{\mathbf{H}}$ |

$\underset{\sim}{\epsilon}$, $\underset{\sim}{e}$ deformation tensor, $\underset{\sim}{\mathbf{F}}$ transformation gradient, $T$ temperature, $V$ electric potential, $\underline{\mathbf{A}}$ potential vector, $\underset{\sim}{\sigma}$ Cauchy stress tensor, $\underset{\sim}{\mathbf{S}}$ first Piola–Kirchoff stress tensor, $\underset{\sim}{\kappa}$ curvature tensor, $\underset{\sim}{\mu}$ couple stress tensor, $H$ enthalpy, $\underline{q}$ heat flux, $\underline{\mathbf{E}}$ electric field, $\underline{\mathbf{H}}$ magnetic field.

**Table 1.** *Some example of* `primal/dual` *variables*

### 5.2.1. *Interface. Definition of a Behavior*

A behavior is in charge of the computation, using as input a *primal* variable (`primal`), of the dual (`dual`) variable as well as the behavior "stiffness matrix" which can be formally expressed as:

$$\frac{\partial\Delta\mathtt{dual}}{\partial\Delta\mathtt{primal}}$$

Examples of `primal/dual` variables are shown in Table 1.

The state of the material is defined by a set of internal variables $\mathcal{V}_{\mathrm{int}}$. They are, for instance, the elastic deformation tensor or the cumulated plastic strain. However, auxiliary variables $\mathcal{V}_{\mathrm{aux}}$, such as the plastic deformation tensors or the stress triaxiality, can also be defined for post–processing purposes or to simplify the calculations. Finally, the user may wish to impose some external parameters `EP` such as the temperature (in a mechanical computation), the grain size, etc. The behavior is expressed using coefficient (`CO`) such as the Young's modulus or hardening parameters. Coefficients are allowed to depend on the set ($\mathcal{V}_{\mathrm{int}}, \mathcal{V}_{\mathrm{aux}}, \mathtt{EP}$) (Figure 9). The material behavior is used to make time increments $t \to t + \Delta t = t'$. The behavior interface can be expressed as follows, in the most general case:

$$\begin{tikzcd}
\boxed{\begin{array}{c}\textbf{Input}\\ \hline \mathtt{primal}^t,\ \Delta\mathtt{primal}\\ \mathtt{dual}^t\\ \mathcal{V}_{\mathrm{int}}{}^t,\ \mathcal{V}_{\mathrm{aux}}{}^t\\ \mathtt{EP}^0,\ \mathtt{EP}^t,\ \mathtt{EP}^{t'}\end{array}} \arrow[r, "\text{behavior}"] & \boxed{\begin{array}{c}\textbf{Output}\\ \hline \mathtt{dual}^{t'}\\ \mathcal{V}_{\mathrm{int}}{}^{t'},\ \mathcal{V}_{\mathrm{aux}}{}^{t'}\\ \dfrac{\partial\Delta\mathtt{dual}}{\partial\Delta\mathtt{primal}}\end{array}}
\end{tikzcd}$$

The task of the behavior is therefore to compute the updated `dual`, $\mathcal{V}_{\mathrm{int}}$ and $\mathcal{V}_{\mathrm{aux}}$ quantities for a given value of the increment of the `primal` variables.

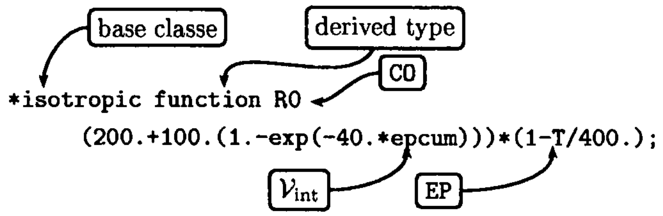

base classe
derived type
CO
*isotropic function R0
(200.+100.(1.-exp(-40.*epcum))*(1-T/400.);
$\mathcal{V}_{\text{int}}$
EP

**Figure 9.** *Declaration of an isotropic hardening variable (base object* ISOTROPIC*) corresponding to the derived type* function*. This object requires a coefficient named* R0*. In this particular example, the coefficient depends on an internal variable* epcum *(cumulated plastic strain) and on an external parameter* T *(temperature)*

External parameters are fully determined. It also has to compute an estimate of the tangent matrix.

The material constitutive equations are usually a differential equations system on the internal variables:

$$\frac{d\mathcal{V}_{\text{int}}}{dt} = \mathcal{F}\left(\mathcal{V}_{\text{int}}, \mathcal{V}_{\text{aux}}, \text{EP}, \texttt{primal}, \frac{d\texttt{primal}}{dt}\right) \qquad [1]$$

The $\texttt{dual}^{t'}$ variable is computed after integration of the previous equation. Integration can be straightforward, as in the case of thermo–linear elasticity; in most cases numerical integration has to be performed. An explicit scheme (Runge–Kutta method) or an implicit scheme ($\theta$–method/mid–point method) can be used. in the second case, equation 1 has to be rewritten incrementally as:

$$\Delta\mathcal{V}_{\text{int}} - \mathcal{G}\left({\mathcal{V}_{\text{int}}}^{\theta}, {\mathcal{V}_{\text{aux}}}^{\theta}, \text{EP}^{\theta}; \texttt{primal}^{\theta}, \Delta\texttt{primal}, \Delta t\right) = 0 \qquad [2]$$

$\theta$ lies between 0 and 1 allowing to go from a explicit Euler scheme ($\theta = 0$) to a fully implicit scheme ($\theta = 1$). $x^{\theta}$ is defined as $x^t + \theta\Delta x$. Equation 2 has then to be solved with respect to $\Delta\mathcal{V}_{\text{int}}$. This is usually achieved using a Newton–Raphson method which requires the calculation of the Jacobian:

$$1 - \frac{\partial\mathcal{G}}{\partial\Delta\mathcal{V}_{\text{int}}} \qquad [3]$$

### 5.2.2. *Implementation*

The element objects were representative of a programmation method strongly based on *factorization*. On the other hand, behaviors are essentially implemented using the *composite* and *association* pattern. Behaviors are build as a dynamic assembly of object of type MATERIAL_PIECE; this object follows the *composite* pattern (Figure 10). Scalar, vectorial and tensorial primal, dual,

$\mathcal{V}_{\text{int}}$ and $\mathcal{V}_{\text{aux}}$ variables are represented by objects: SCALAR_PRIMAL, SCALAR_DUAL, TENSOR2_VINT, etc. These are data members of a MATERIAL_PIECE object which is part of a *composite*; using the recursive tree associated with the *composite* pattern, the variable can then be attached to the object controling the all tree (usually a BEHAVIOR object). This mechanism, which locally associates a MATERIAL_PIECE and its variables and builds the *composite* tree, allows to dynamically create behaviors.

Figure 10 illustrates this mechanism in the case of plastic/viscoplastic behaviors (GEN_EVP)[1] with various inelastic mechanisms including both isotropic and kinematic hardening [ CON 89, SAÏ 95, FOE 97]. In this example, the object describing kinematic hardening (KINEMATIC) holds an internal variable $\underset{\sim}{\alpha}$ of type TENSOR2_VINT. The object describing isotropic hardening (ISOTROPIC) has a scalar internal variable (SCALAR_VINT). When adding to an inelastic dissipation potential (POTENTIAL), a new kinematic hardening mechanism, the local variable $\underset{\sim}{\alpha}$ is automatically linked to the potential and then to the behavior. Memory allocation, output management, etc., are therefore completely handled by the base object MATERIAL_PIECE.

The integration of the constitutive equations (Equation 1 or 2) has then to be handled. The developer can use objects (RUNGE_KUTTA and THETA_METHOD) managing both numerical integration scheme (explicit/implicit). Behaviors can derive for one or both of these objects; multiple inheritance is then used. They declare virtual functions whose task is to compute Equation1 or 2, and 3. In the example shown on Figure 10, the set GEN_EVP+POTENTIAL (i.e., behavior/deformation mechanism) builds an *association*. An object of type GEN_EVP can control several POTENTIAL objects. KINEMATIC and ISOTROPIC are, on the other hand, used using interface functions. A potential may have one or more kinematic hardening variables but only one isotropic hardening variable. There are other objects used to describe the behavior. FLOW specifies the type of flow (plastic, Norton, etc.). CRITERION specifies the yield surface; Von Mises is used as default. These objects use the *object factory* pattern to create the different derived types: linear or non–linear, with or without recovery, etc.

Coefficients are an example of an encapsulated hierarchy. The base class COEFFICIENT defines an interface allowing the computing of the value of the coefficient and its derivative with respect to the different variables. The derived types include coefficient which are constant, defined by a table, defined by a function or random. The COEFF object is used as a container class to encapsulate the whole hierarchy. MATERIAL_PIECE usually have COEFF data members.

## 5.3. *Other Objects*

At the present time, Zébulon contains about 800 objects (not including the different template classes). Most of them are however, part of the 50 object hierarchies following the *object factory* pattern. The principal base objects are:

[1]Generic elasto—viscoplastic

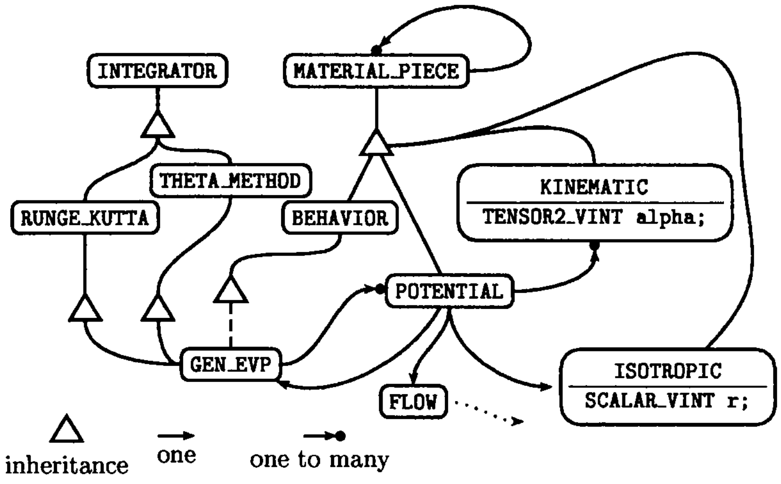
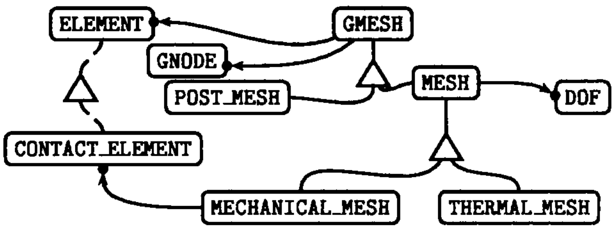


**Figure 10.** *OMT diagram* [ BOO 97] *showing the organization of some of the different objects using to build material behaviors. Intermediate objects between* BEHAVIOR *and* GEN_EVP *are used to factorize functionalities describing thermo-elastic properties. The* FLOW *object also derives from* MATERIAL_PIECE

GMESH. This object represents meshes. A hierarchy has been created in order to separate the tasks related to the geometry (GMESH), to the dofs (MESH), or to the post-processing (POST_MESH), etc.:



Specialized meshes (MECHANICAL_MESH and THERMAL_MESH) can perform special tasks. For instance, MECHANICAL_MESH can create and store contact elements.

BC. This object handles Boundary Conditions.

RELATIONSHIP. This object handles linear relationship between dofs.

BASE_PROBLEM. This object is the common base from all problems. These include the FE calculations, but also optimization [ LER 97, BES 98], pre– and post–processing. Its user interface has three principal functions: initialization, verification and execution. PROBLEM is the base for all FE problems. It contains a mesh (MESH), a list of boundary conditions and relationships and a resolution algorithm (ALGORITHM).

ALGORITHM. The ALGORITHM hierarchy contains resolution algorithms associated with the different FE problems. In the case of mechanical problems Newton, BFGS [ BAT 82] and Riks [ RIK 79] methods have been implemented.

GLOBAL_MATRIX. This object handles the storage and assembly of the elementary stiffness matrices $[K_e]$. Derived objects can solve linear systems $[K]\{U\} = \{R\}$. Frontal and skyline methods have been implemented.

## 6. Concluding Remarks

The present work has attempted to propose design principles of object–oriented finite element codes, particularly in the case where the project is large and the desired code is to be a flexible general purpose tool. The following topics have been discussed:

**Base library.** A basic library must be used to handle the bulk of the mathematical operations, the repetitive input/output tasks, string manipulation and storage. Templates are powerful tools which simplifies programming. The proposed library introduces a tensor object which will greatly aid the programming of complex constitutive equations.

**Patterns.** Patterns are design methodologies which help to hve a consistent development strategy. Defining patterns appears to be an important task as they tend to remain stable while the code organization may change rapidly. This work introduces some design patterns which intend to ease extensibility and flexibility.

**Numerical efficiency.** Efficiency has been shown to be good when using C++ compared to other OOLs [ DUB 93, BES 97]. However, better results are still obtained using FORTRAN [ HAN 94]. This is due to both the fact that FORTRAN is highly optimized and to abstraction which requires "dynamic linking". Small objects, which are often created and destroyed, must be implemented carefully. In some cases, it is worth losing numerical efficiency in order to obtain a simple and safe implementation. The plane stress elements, presented in the work, require more dofs (an consequently an increased computational effort) than is required using a standard treatment of plane stress conditions. However, this permits avoiding of treating "special cases" while implementing material behaviors which clearly allows obtaining a safer extensible code.

**A language dedicated to the FEM.** No language can claim to be specifically dedicated the to FEM. Although FORTRAN 77 still seems to be the preferred language for scientific programming, this is principally due a long practice and a high numerical efficiency. By itself, the language syntax does not ease scientific programming as it does not easily allow handling quantities such as vectors, tensors, nodes, meshes, etc. FORTRAN 90 and Ada now implement these functionalities. Object–oriented languages are not dedicated to the FEM. However, they naturally allow defining objects whose interfaces form a new programming syntax. The results is a new "macro–language" which is used for new developments (see Figure 7).

## 7. References

[ AAZ 93] K. AAZIZOU, J. BESSON, G. CAILLETAUD, F. HOURLIER. "Une approche C++ du calcul par éléments finis". In *Colloques national en calcul des structures*, Giens, France, May 11-14, 1993. Hermès, Paris.

[ BAT 82] K.J. BATHE. "*Finite element procedures in engineering analysis*". Prentice Hall, Inc., 1982.

[ BAT 91] J.L. BATOZ G. GHATT. "*Modélisation des structures par éléments finis, I—III*". Hermès, 1991.

[ BES 88] J.-M. BESSON, M. JAEGER, O. DEBORDES. "Homogénéisation de composites à base de tissus de fibres". In D. GAY C BORD, editors, *Composite Structures*, pages 77–90, 20–22 June 1988.

[ BES 97] J. BESSON R. FOERCH. "Large Scale Object–Oriented Finite Element Code Design". *Computer Methods in Applied Mechanics and Engineering*, 142:165–187, 1997.

[ BES 98] J. BESSON, R. LE RICHE, R. FOERCH, G. CAILLETAUD. "Application of object–oriented programming techniques to the finite element method. Part II— Application to material Behaviors". *Revue europénne des éléments finis*, to be published, 1998.

[ BET 96] B.P. BETTIG R.P.S HAN. "An object–oriented framework for interactive numerical analysis in a graphical user interface environment". *Int. J. for numerical methods in engineering*, 39:2945–2971, 1996.

[ BOO 97] G. BOOCH. "*Des solutions objets*". International Thomson Publishing France, Paris, 1997.

[ BRE 92] P. BREITKOPF G. TOUZOT. "Architecture des logiciels et langages de modélisation". *Revue Européenne des éléments finis*, 1(3):333–368, 1992.

[ CAR 94] A. CARDONA, I. KLAPKA, M. GERADIN. "Design of a new finite element programming environment". *Engineering computations*, 11:365–381, 1994.

[ COM 96] S. COMMEND T. ZIMMERMANN. "Finite element preprocessing with Java". Technical Report, EPFL, 1996.

[ CON 89] E. CONTESTI G. CAILLETAUD. "Description of creep-plasticity interaction with non-unified constitutive equations". *Nuclear Eng. and Design*, 116:265, 1989.

[ DEB 91] R. DE BORST. "Simulation of strain localization: a reappraisal of the Cosserat continuum". *Eng. Computations*, 8:317–332, 1991.

[ DEL 85] H.G. DELORENZI. "Energy release rate calculations by the finite element method". *Engineering Fracture Mechanics*, 1985.

[ DUB 92a] Y. DUBOIS-PÉLERIN, T. ZIMMERMANN, P. BOMME. "Object-oriented finite element programming: II. A prototype program in Smalltalk". *Computer Methods in Applied Mechanics and Engineering*, 98:361–397, 1992.

[ DUB 92b] Y.-D. DUBOIS-PÉLERIN. "*Object-oriented finite elements: programming concepts and implementation*". PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 1992.

[ DUB 93] Y. DUBOIS-PÉLERIN T. ZIMMERMANN. "Object-oriented finite element programming: III. An efficient implementation in C++". *Computer Methods in Applied Mechanics and Engineering*, 108:165–183, 1993.

[ FOE 97] R. FOERCH, J. BESSON, G. CAILLETAUD, P. PILVIN. "Polymorphic Constitutive Equations in Finite Element Codes". *Computer Methods in Applied Mechanics and Engineering*, 141:355–372, 1997.

[ FOR 90] B.W.R. FORDE, Foschi R.O., Stiemer S.F.. "Object-Oriented Finite Element Analysis". *Computers & Structures*, 34(3):355–374, 1990.

[ FOR 97] S. FOREST. "Mechanics of Generalized Continua: Construction by Homogeneization". In *40ème Colloque de Métallurgie, Comportement Mécanique et Effets d'Echelle, INSTN.* to appear in Journal de Physique IV., June 24-26, 1997.

[ GAJ 96] R.R. GAJEWSKI T. KOWALCZYK. "A prototype object-oriented finite element method program: class hierarchy and graphic user interface". *Computer assisted mechanics and engineering science*, 3:65–74, 1996.

[ GAM 94] E. GAMMA, R. HELM, R. JOHNSON, J. VLISSIDES. "*Design Patterns: elements of reusable object-oriented software*". Addison Wesley professional computing series, 1994.

[ HAN 94] S.W. HANEY. "Is C++ fast enough for scientific computing?". *Computers in physics*, 8(6), 1994.

[ HUG 80] J.R. HUGHES. "Generalization of selective integration procedures to anisotropic and non linear media". *Int. J. Numerical Methods in Engineering*, 15:1413–1418, 1980.

[ KON 95] X.A. KONG D.P. CHEN. "An object-oriented design of FEM programs". *Computers & Structures*, 57:157–166, 1995.

[ LAF 97] C. LAFFRA. "*Advanced Java*". Prentice Hall PTR, 1997.

[ LER 97] L. LE RICHE, F. FEYEL, J. BESSON, G. CAILLETAUD, M. GUTMANN, R. FOERCH. "L'objet matériau, de l'identification au calcul de structures". In J.P. Pelle B. PESEUX, D. Aubry M. TOURATIER, editors, *Actes du*

*Troisième Colloque National en Calcul des Structures*, pages 583–588. Presses Académiques de l'Ouest, 20–23 Mai 1997 1997.

[ LUC 92] D. LUCAS, B. DRESSLER, D. AUBRY. "Object–oriented finite element programming using the ADA language". In Ch. Hirsch et AL., editor, *Numerical Methods in Engineering'92*, pages 591–598. Elsevier Science Publishers B.V., 1992.

[ LUC 94] D. LUCAS. "*Méthode des éléments finis et programmation orientée objet. Utilisation du langage Ada*". PhD thesis, Ecole Centrale de Paris, 1994.

[ MAC 92] R.I. MACKIE. "Object–oriented programming of the finite element method". *Int. J. for Numerical Methods in Engineering*, 35:424–436, 1992.

[ MEH 97] K. MEHLHORN, S. NÄHER, C UHRIG. "*The LEDA User Manual, version 3.5.1*", 1997.

[ MEN 93] P. MENÉTREY T. ZIMMERMANN. "Object–Oriented Non-Linear Finite Element Analysis: Application to J2 Plasticity". *Computers & Structures*, 49:767–777, 1993.

[ MIL 88] G.R. MILLER. "A LISP-Based Object-Oriented Approach to Structural Analysis". *Engineering with Computers*, 4:197–203, 1988.

[ PAR 74] D.M. PARKS. "A stiffness derivative finite element technique for determination of crack tip stress intensity factors". *Int. J. Fracture*, 10(4):487–502, 1974.

[ RIC 68] J.R. RICE. "A path independant integral and the approximate analysis of strain concentration by notched and cracks". *J. Appl. Mech.*, 35:379, 1968.

[ RIK 79] E. RIKS. "An incremental approach to the solution of snapping and buckling problems". *Int. J. Solids Structures*, 15:529–551, 1979.

[ RUM 91] J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY, W. LORENSEN. "*Object-Oriented Modeling and Design*". Prentice Hall, New Jersey, 1991.

[ SAÏ 95] K. SAÏ G. CAILLETAUD. "Study of Plastic/Viscoplastic Models with Various Inelastic Mechanisms". *Int J. Plasticity*, 11(8):991–1005, 1995.

[ SCH 92] S.-P. SCHOLZ. "Elements of an Object-Oriented FEM++ program in C++". *Computers & Structures*, 43:517–529, 1992.

[ SIM 85] J.C. SIMO R.L. TAYLOR. "Consistent tangent operators for rate–independent elastoplasticity". *Computer Methods in Applied Mechanics and Engineering*, 48:101–118, 1985.

[ STR 86] B. STROUSTRUP. "*The C++ programming language*". Addison-Wesley Publishing, Reading, Mas., 1986.

[ STR 88] B. STROUSTRUP. "What is object–oriented programming ?". *IEEE Software*, 5:10–20, 1988.

[ STR 94] B. STROUSTRUP. "*The Design and Evolution of C++*". Addison-Wesley Publishing Company, Inc., Reading, Mass, 1994.

[ VER 88] P. VERPEAUX, T. CHARRAS, A. MILLARD. "CASTEM 2000: une approche moderne du calcul des structures". In J.M. FOUET, P. LADEVÈZE, R. OYAHON, editors, *Conférence, Calcul des Structures et Intelligence Artificielle*, volume 2. Pluralis, 1988.

[ WAL 96] L. WALTERTHUM. *"Programmation orientée objet et calculs par éléments finis. Application à la conception d'un logiciel de simulation en mise en forme des matériaux"*. PhD thesis, Université de Franche-Comté, 1996.

[ ZIM 92] T. ZIMMERMANN, Y. DUBOIS-PÉLERIN, P. BOMME. "Object-oriented finite element programming: I. Governing principles". *Computer Methods in Applied Mechanics and Engineering*, 98:291–303, 1992.

## Appendix

## A Basic tools

Basic tools used in Zébulon are briefly described. They include character string, files, different mathematical objects such as matrices, vectors, second–order tensors, square matrices and generic types such as arrays, encapsulated pointers, lists, dictionaries, etc.

Similar objects can be found elsewhere. The **Standard Template Library (STL)** can be used. Free packages, such as the LEDA [ MEH 97] library, can also be used. These libraries are not used in Zébulon. The STL was not available when the code started to be redesigned. The LEDA library does not include the possibility to define sub–matrices and sub–vectors which are widely used in Zébulon. Second–order tensors are also missing.

### A1. *Run time verification*

The code can easily be debugged at run time using the `assert` C instruction. However, verification can be time consuming, and should consequently only be used for debugging purposes. For instance it is possible to check array overflow:

```
VECTOR v(4); // vector of size 4
v[4]=0.;     // run time error (overflow)
```

Note that FORTRAN 90 and Java automatically check for overflow.

### A2. *Encapsulation of pointers*

In C++, like in C, handling of pointers is difficult as they are not automatically destroyed (absence of a garbage collecting mechanism). Memory allocation must however be handled carefully, as very large objects (as big as a whole FE problem) can be dynamically created and destroyed. The template class `PTR<T>` helps solving this problem as it encapsulates a pointer of type `T*` which is automatically freed as the destructor of the encapsulating object is called:

```
void f()
{  OBJ* p_obj = new OBJ;

}
```

Allocated memory is not freed. Erroneous code.

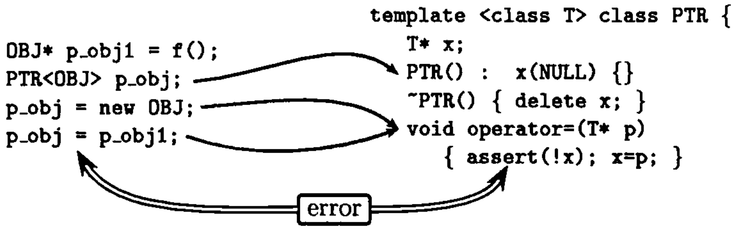
```
void f()
{  OBJ* p_obj = new OBJ;
   delete p_obj;
}
```

Allocated memory is freed. Correct code.

```
void f()
{  PTR<OBJ> p_obj = new OBJ;
}
```

Correct code as the destructor of PTR<> frees the encapsulated pointer

Run time verification can also be used to avoid loosing addresses. Use of standard pointers is limited to cases involving the use of objects but not their actual storage.



```
OBJ* p_obj1 = f();
PTR<OBJ> p_obj;
p_obj = new OBJ;
p_obj = p_obj1;
```

```
template <class T> class PTR {
   T* x;
   PTR() :  x(NULL) {}
   ~PTR() { delete x; }
   void operator=(T* p)
      { assert(!x); x=p; }
```

## A3. *Arrays and lists*

Arrays (ARRAY<>) and lists (LIST<>) (i.e., arrays that can grow and shrink) are implemented as templates. Lists are frequently implemented as chained lists; however, in the case of the FEM, lists are, in most cases, created and filled during the creation of the problem. During resolution, they are used as arrays of fixed size. In order to avoid using the chained list mechanism to get a given item, lists derive from arrays. Adding or removing items can be done by resizing the storage zone (with some buffer mechanism). The following inheritance hierarchy is then obtained:

$$\begin{tikzcd} \text{ARRAY<>} & \text{CARRAY<>} \arrow[l] & \text{LIST<>} \arrow[l] \end{tikzcd}$$

where CARRAY<> is a class representing arrays of objects for which the == operator is defined. Arrays of pointers can be defined as ARRAY<OBJ*> or ARRAY< PTR<OBJ> >. For instance, the mesh object MESH handles the storage of elements. To easily perform this task, it uses the class ARRAY< PTR<ELEMENT> > so that elements are destroyed when the mesh is itself destroyed. On the other hand, the node objects contain the table of the elements to which they belong. Storage of the corresponding addresses is done using the class ARRAY<ELEMENT*>.

## A4. *Mathematical objects*

One of the many advantages of C++, which is immediately perceived by the programmer, is the ability to define overloaded operators which allow the writing of clear, dimension independent, formulas such as:

```
TENSOR2 n = 1.5 * deviator(sigma)/mises(sigma);
TENSOR4 N = n^n;
double x = (v|v);
NODE middle = (node1+node2)/2.;
```

$$\underset{\sim}{n} = \frac{3}{2}\frac{\underset{\sim}{s}}{J(\underset{\sim}{\sigma})}$$

$$\underset{\approx}{N} = \underset{\sim}{n} \otimes \underset{\sim}{n}$$

$$x = \underline{v} : \underline{v}$$

Similar objects are often found in different object libraries [ MEH 97, SCH 92]. In addition, Zébulon uses second–order tensors. They are stored using a Voigt notation modified for symmetric tensors:

$$\begin{array}{ll} \text{1D} & (t_{11},\, t_{22},\, t_{33}) \\ \text{2D} & \left(t_{11},\, t_{22},\, t_{33},\, \sqrt{2}t_{12}\right) \\ \text{3D} & \left(t_{11},\, t_{22},\, t_{33},\, \sqrt{2}t_{12},\, \sqrt{2}t_{23},\, \sqrt{2}t_{31}\right) \end{array}$$

Use of such objects allows, for instance, the implementation pf material constitutive equations and elements without having to explicitly deal with the space dimension. It is, however, necessary to carefully implement these small objects which are widely used in the code. For instance, the following code:

```
VECTOR v=v1+v2+v3+v4;
```

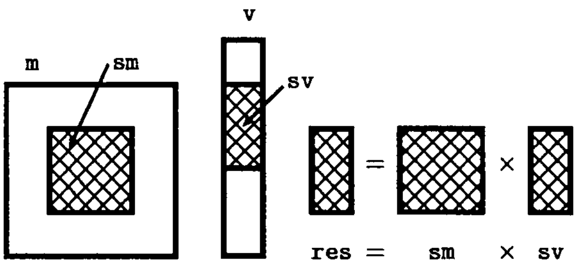
can be much less efficient than

```
VECTOR v=v1; v+=v2; v+=v3; v+=v4;
```

due to the excessive number of copies and destruction requiring dynamic memory allocation. Indeed, the first solution should always been used as it achieves the higher readability. These difficulties can, however, be overcome using techniques such as reference counting and by explicitly handling memory allocation during the creation and the destruction of these objects. This is indeed very important in the case of small objects, such as second–order tensors, as many instances are created, initialized and destroyed [ HAN 94].

Zébulon also allows the definition of sub–matrices and sub–vectors. For instance:

| | |
|---|---|
| `MATRIX m(6,6);` | matrix of size $6 \times 6$ |
| `VECTOR v(12);` | vector of size 12 |
| `MATRIX sm(3,3,m,1,1);` | sub–matrix of size $3 \times 3$ starting at position (1,1) |
| `VECTOR sv(3,v,2);` | sub–vector of size 3 starting at position (2) |
| `VECTOR res = sm*sv;` | sub–matrix$\times$sub–vector product |



## A5. *Other objects*

The code also contains objects such as character strings (STRING) and file objects used to manage input/output (ASCII_FILE, ZFSTREAM, ZIFSTREAM and ZOFSTREAM). ZFSTREAM files behave like standard C++ fstream objects and allow the handling of input/output in the case of parallel computations.

Lists indexed by other objects (dictionary) are also used (DICT<S,K>) [ MEH 97]. For instance, sets of nodes (NSET) are stored in the GMESH object as a dictionary indexed by strings (STRING).

```
class GMESH { ...
   DICT< PTR<NSET>, STRING > nset;
... };
nset["bottom"] = ...;
```

The code also has numerical integration methods (Runge–Kutta, $\theta$–method), methods to find roots of functions (Newton–Raphson) encapsulated in objects.