# Rationale of the DUNE Grid Module Interface

Peter Bastian

February 20, 2003

**Abstract**

This document describes how static polymorphism might be used in the case where a set of related classes must be used consistently to implement a common interface. The solution uses the Barton-Nackman trick and introduces a new concept, the bonds class. This concept is then applied to describe the DUNE grid module interface. The main emphasis here is to provide a general scheme to implement a set of related classes. The members of the individual classes are still subject to discussion.

## 1 Static Polymorphism

In *dynamic* polymorphism we use interface base classes having only pure virtual methods to describe a general interface. Concrete classes are derived publicly from that interface and provide implementations of the methods.

Algorithms are programmed using the abstract interface. The algorithms are used by calling them with concrete objects from derived classes. This technique is efficient if the code of the methods is sufficiently large. For fine granular interfaces, however, virtual function overhead is not tolerable.

A solution to that problem, used extensively e. g. in the standard template library, is to use *static* polymorphism implemented with templates in C++. The idea is to parametrize the algorithm with a class at compile-time. Methods can now be inlined and no virtual method dispatch is needed.

### 1.1 The Barton-Nackman Trick

A clever way to implement static polymorphism was presented in the book [BN94].

Consider the following code:

```
1  template<class Engine>
2  class Base {
3  public:
4      int begin () {
5          return asEngine().begin();
6      }
7  private:
8      Engine& asEngine() {
9          return static_cast<Engine&>(*this);
10     }
11 };
12
13 class A : public Base<A>
14 {
15 public:
16     int begin () {return 23;}
17 };
18
19 class B : public Base<B>
20 {
21 public:
22     int begin () {return 14;}
23 };
```

```
24
25 template<class Engine>
26 void foo (Base<Engine>& x)
27 {
28     int i=x.begin();
29 }
30
31 int main ()
32 {
33     A a; foo(a);
34     B b; foo(b);
35 }
```

Classes A and B are supposed to be two different implementations of the same interface. Sometimes such classes are called *engines*. A first idea would be to parametrize a function, such as foo, directly with either A or B. The problem with that is that A and B are totally unrelated classes. The compiler cannot check that A and B really implement the same interface.

The Barton-Nackman trick achieves this by deriving A and B from a common base class Base. In order to allow function inlining by the compiler, the derived class is a template parameter to the base class. In the base class all methods are forwarded to the template parameter which is known at compile-time and methods can be inlined.

The generic function foo is called with a derived class object (which is possibly due to public derivation) in main() but the implementation of foo uses the (parametrized) base class.

## 1.2 Bonds

In a real situation a module is implemented by a set of related classes. E. g. a grid manager module migh have classes for the grid, vertices, elements, several iterators to run over the data structure, classes for reporting errors and so on. A natural way to describe such an interface is through nested classes within some enclosing class representing the whole interface. Unfortunately it turns out that the Barton-Nackman trick is difficult to extend to nested classes.

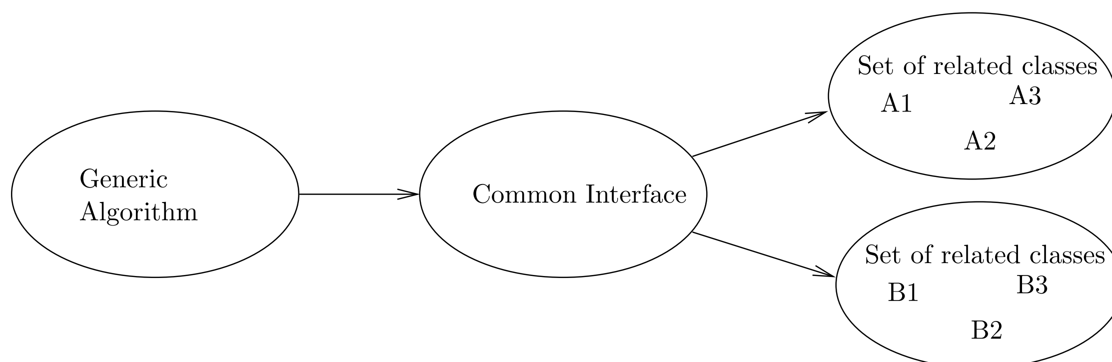


Figure 1: Generic Algorithm using different implementations of a common interface.

Therefore we go a different way here and keep the related types as seperate classes not in the scope of an enclosing class (a namespace should be used for that purpose). An additional problem is that the related classes mutually may use each other as arguments and return types. An implementation with static polymorphism must ensure that only corresponding classes are used together.

Here is our set of related classes A1, A2 and A3 which are supposed to implement an interface of a certain module:

```
1 // forward declaration of classes
2 // because they mutually use each other
3 class A1;
4 class A2;
5 class A3;
6
7 // combine related types in a single class which
8 // will be used as template parameter
```

```
9  // This is to make the names of the implementation
10 // side known on the usage side
11 class ABonds {
12 public:
13     typedef A1 T1;
14     typedef A2 T2;
15     typedef A3 T3;
16 };
17
18 // Class declarations.
19 // Use Barton-Nackman trick to achieve public derivation
20 // from base and static polymorphism
21 class A1 : public Base1<A1, ABonds>
22 {
23 public:
24     typedef ABonds Bonds;
25     A2 begin ();
26 };
27
28 class A2 : public Base2<A2, ABonds>
29 {
30 public:
31     typedef ABonds Bonds;
32     A3 begin ();
33 };
34
35 class A3 : public Base3<A3, ABonds>
36 {
37 public:
38     typedef ABonds Bonds;
39     A1 begin ();
40 };
41
42 // inline implementation of methods
43 inline A2 A1::begin () {cout << "A1::begin() called" << endl; return A2();}
44 inline A3 A2::begin () {cout << "A2::begin() called" << endl; return A3();}
45 inline A1 A3::begin () {cout << "A3::begin() called" << endl; return A1();}
```

Note how methods of the three classes mutually return each other as return types. In order to make the names of all related classes known on the user side of the interface we introduce a new class called ABonds which collects the names of all related classes and maps them to universal names T1, T2, . . . , used in the interface. Each of the three classes A1, . . . , A3 defines the name Bonds in its scope which now can be used in the base classes and on the user side of the interface to deduce the names of the other related classes.

Here is the definition of the common interface using the Barton-Nackman trick:

```
1  // Module interface described by a set of related base classes.
2  // These classes mutually use each other as arguments and
3  // return types.
4
5  template<class Engine1, class Bonds = typename Engine1::Bonds>
6  class Base1 {
7  public:
8      typename Bonds::T2 begin () {
9          return asEngine1().begin();
10     }
11 private:
12     Engine1& asEngine1() {
13         return static_cast<Engine1&>(*this);
14     }
15 };
16
17 template<class Engine2, class Bonds = typename Engine2::Bonds>
18 class Base2 {
```

```
19 public:
20     typename Bonds::T3 begin () {
21         return asEngine2().begin();
22     }
23 private:
24     Engine2& asEngine2() {
25         return static_cast<Engine2&>(*this);
26     }
27 };
28
29 template<class Engine3, class Bonds = typename Engine3::Bonds>
30 class Base3 {
31 public:
32     typename Bonds::T1 begin () {
33         return asEngine3().begin();
34     }
35 private:
36     Engine3& asEngine3() {
37         return static_cast<Engine3&>(*this);
38     }
39 };
```

Each base class now has two template parameters, the engine giving the implementation of the class and a bonds class containing the names of the other related types. The bonds class is supplied when publicly deriving from the class. If the base classed is, however, used in a function call on the user side, the bonds class is automatically deduced from the first argument and need not be specified.

Finally, here is some code using these classes:

```
1  #include<iostream>
2
3  using namespace std;
4
5  #include"interface.hh"
6  #include"implementationA.hh"
7  #include"implementationB.hh"
8
9  // two generic algorithms
10 template<class Engine3>
11 int bar (Base3<Engine3>& x3)
12 {
13     return 17+4;
14 }
15
16 template<class Engine1, class Engine2>
17 void foo (Base1<Engine1>& x1, Base2<Engine2>& x2)
18 {
19     typedef typename Engine1::Bonds Bonds;
20
21     typename Bonds::T2 i = x1.begin();
22     typename Bonds::T3 j = x2.begin();
23
24     int k=bar(j);
25 }
26
27 int main ()
28 {
29     // create objects of concrete classes
30     A1 a1; A2 a2; A3 a3;
31     B1 b1; B2 b2; B3 b3;
32
33     // use abstract interface in algorithm
34     foo(a1,a2);
35     foo(b1,b2);
```

```
}
```

Note how the different types are used in the functions foo and bar. Derived class objects are used through the base class interface. Thus it can be ensured that only methods implemented in the interface are used. However, if new objects are required, e. g. as return values, derived class (concrete) object are created.

# 2 Application to the Grid Module Interface

## 2.1 Overview

The grid interface consists of the following five class templates:

1. Grid<dim>. A grid, as a first approximation, is a container holding elements. Our grids are assumed to be hierarchic (subdivided into levels) and nested (i. e. each coarse grid element is covered exactly by a subset of fine grid elements). Do we want to have this more general?

2. Element<codim,dim>. Elements are parametrized by dimension and codimension. An element of dimension $d$ and codimension $c$ is a $d-c$-dimensional object. The interface is quite fat (i. e. each element provides a lot of functionality that needed not be implemented in all cases) and uniform (means that nearly all elements should have the same interface regardless of dimension and codimension). Elements of codimension $c=d$ (i. e. vertices) are different from elements with $c<d$ and are defined by partial specialization. The remaining types are used to iterate over and access elements.

3. ElementPtr<codim,dim>. This template is used to reference elements. It is similar to an iterator but without the increment and decrement methods. Elements provide access to other elements through this type in order to represent the topology of the mesh.

4. LevelIterator<codim,dim>. Used to iterate over elements of a certain dimension and codimension. Begin and end are provided by the grid.

5. HierarchicIterator<codim,dim>. Allows one to iterate over the element tree starting at a given element. The tree is provided for all elements with codimension smaller than dimension (i. e. not for vertices). Thus one can hierarchically loop over the subdivisions of a given depth of a given element, face or edge in 3D.

## 2.2 Usage

We start with a main program showing the use of the interface. Unfortunately, g++ (in no version) is not able to compile the code because member templates combined with function templates do not work properly. The Intel compiler runs fine.

Here is a little program creating a mesh and calling a generic algorithm that loops over elements of codimension 1.

```
#include<iostream>

using namespace std;

#include"grid.hh"
#include"sgrid.hh"

template<class Engine, int codim>
void bar (LevelIterator<Engine,codim>& i)
{

}

template<class Engine>
void foo (Engine& g)
{
    // loop over elements of codim 1 of level 1
    for (typename LevelIterator<Engine,1>::Engine i = g.lbegin<1>(1); i!=g.lend<1>(1); i++)
```

```
19          cout << i->level();
20
21      // make element of codim 0
22      Element<Engine,0> e;
23  }
24
25  int main ()
26  {
27      // make object of concrete class
28      SGrid<1> sgrid(1.0,2,2);
29
30      // call generic function
31      foo(sgrid);
32
33      // test generic cube grid mapper
34      Tupel<int,2> N;
35      N[0] = 4; N[1] = 3; // N[2] = 10; N[3] = 10;
36
37      CubeMapper<2> cm(N);
38
39      Tupel<int,2> z;
40      for (int i1=0; i1<=2*N[1]; i1++)
41          for (int i0=0; i0<=2*N[0]; i0++)
42          {
43              z[0]=i0; z[1]=i1;
44              cout << "␣␣[" << i0 << "," << i1 << "]"
45                   << "␣c␣=␣" << cm.codim(z)
46                   << "␣n␣=␣" << cm.n(z)
47                   << endl;
48          }
49  }
```

The implementation of the interface in sgrid.hh provides a general structured (sequential) mesh. Implementation is not completed yet. It will provide all iterators, will work in all dimensions (also $d > 3$ !) and uses virtualized elements (i. e. space requirements do not depend on the grid size).

## 2.3 The Interface

```
1   #ifndef __GRID_HH__
2   #define __GRID_HH__
3
4   //**********************************************************************
5   //**********************************************************************
6   // Dune General Grid Interface
7   //**********************************************************************
8   //**********************************************************************
9
10  // map from Bonds to parametrized types
11  // maps are implemented for dimension 1,2,3, more could be added !
12
13  // here: Element
14  template<typename Bonds, int codim, int dim>
15  struct ElementEngine {
16  };
17
18  template<typename Bonds, int dim>
19  struct ElementEngine<Bonds,dim,dim> {
20      typedef typename Bonds::Vertex Engine;
21  };
22
23  template<typename Bonds, int dim>
24  struct ElementEngine<Bonds,0,dim> {
```

```
25      typedef typename Bonds::Element Engine;
26  };
27
28  template<typename Bonds>
29  struct ElementEngine<Bonds,1,2> {
30      typedef typename Bonds::Edge Engine;
31  };
32
33  template<typename Bonds>
34  struct ElementEngine<Bonds,1,3> {
35      typedef typename Bonds::Face Engine;
36  };
37
38  template<typename Bonds>
39  struct ElementEngine<Bonds,2,3> {
40      typedef typename Bonds::Edge Engine;
41  };
42
43  // here: ElementPtr
44  template<typename Bonds, int codim, int dim>
45  struct ElementPtrEngine {};
46
47  template<typename Bonds, int dim>
48  struct ElementPtrEngine<Bonds,dim,dim> {
49      typedef typename Bonds::VertexPtr Engine;
50  };
51
52  template<typename Bonds, int dim>
53  struct ElementPtrEngine<Bonds,0,dim> {
54      typedef typename Bonds::ElementPtr Engine;
55  };
56
57  template<typename Bonds>
58  struct ElementPtrEngine<Bonds,1,2> {
59      typedef typename Bonds::EdgePtr Engine;
60  };
61
62  template<typename Bonds>
63  struct ElementPtrEngine<Bonds,1,3> {
64      typedef typename Bonds::FacePtr Engine;
65  };
66
67  template<typename Bonds>
68  struct ElementPtrEngine<Bonds,2,3> {
69      typedef typename Bonds::EdgePtr Engine;
70  };
71
72   // here: LevelIterator
73  template<typename Bonds, int codim, int dim>
74  struct LevelIteratorEngine {};
75
76  template<typename Bonds, int dim>
77  struct LevelIteratorEngine<Bonds,dim,dim> {
78      typedef typename Bonds::VertexLevelIterator Engine;
79  };
80
81  template<typename Bonds, int dim>
82  struct LevelIteratorEngine<Bonds,0,dim> {
83      typedef typename Bonds::ElementLevelIterator Engine;
84  };
85
86  template<typename Bonds>
87  struct LevelIteratorEngine<Bonds,1,2> {
```

```
88      typedef typename Bonds::EdgeLevelIterator Engine;
89  };
90
91  template<typename Bonds>
92  struct LevelIteratorEngine<Bonds,1,3> {
93      typedef typename Bonds::FaceLevelIterator Engine;
94  };
95
96  template<typename Bonds>
97  struct LevelIteratorEngine<Bonds,2,3> {
98      typedef typename Bonds::EdgeLevelIterator Engine;
99  };
100
101 // here: HierarchicIterator
102 template<typename Bonds, int codim, int dim>
103 struct HierarchicIteratorEngine {};
104
105 template<typename Bonds, int dim>
106 struct HierarchicIteratorEngine<Bonds,dim,dim> {
107     typedef typename Bonds::VertexHierarchicIterator Engine;
108 };
109
110 template<typename Bonds, int dim>
111 struct HierarchicIteratorEngine<Bonds,0,dim> {
112     typedef typename Bonds::ElementHierarchicIterator Engine;
113 };
114
115 template<typename Bonds>
116 struct HierarchicIteratorEngine<Bonds,1,2> {
117     typedef typename Bonds::EdgeHierarchicIterator Engine;
118 };
119
120 template<typename Bonds>
121 struct HierarchicIteratorEngine<Bonds,1,3> {
122     typedef typename Bonds::FaceHierarchicIterator Engine;
123 };
124
125 template<typename Bonds>
126 struct HierarchicIteratorEngine<Bonds,2,3> {
127     typedef typename Bonds::EdgeHierarchicIterator Engine;
128 };
129
130 // here: Grid
131 template<typename Bonds, int dim>
132 struct GridEngine {
133     typedef typename Bonds::Grid Engine;
134 };
135
136 //*******************************************************************
137 // Element base class
138 template<int d>
139 class Point {
140 public:
141     Point() {}
142     Point (double* y) {for(int i=0; i<dim; i++) x[i]=y[i];}
143     double& operator[] (int i) {return x[i];}
144 private:
145     double x[d];
146 };
147
148 // Elements use a fat and uniform interface
149 template<typename engine, int codim,
150          int dim = engine::dimension, typename Bonds = typename engine::Bonds>
```

```
151 class Element {
152 public:
153     // NEVER use first template parameter, it is only used for
154     // getting default template parameters in function call
155     typedef typename ElementEngine<Bonds,codim,dim>::Engine Engine; // my engine
156     enum { dimension=dim };      // know your own dimension
157     enum { codimension=codim }; // know your own codimension
158 
159     int level () {return asEngine.level();}
160     int id ()  {return asEngine.id();}
161     int kind() {return asEngine.kind();}
162 
163     // hierarchical access to subelements of higher codimension
164     int subelements () {return asEngine.subelements();}
165     typename ElementPtrEngine<Bonds,codim+1,dim>::Engine subelement (int i)
166         { return asEngine().subelement(i);}
167 
168     // connected elements of codim 0
169     int elements () {return asEngine.elements();}
170     typename ElementPtrEngine<Bonds,0,dim>::Engine element (int i)
171         { return asEngine().element(i);}
172 
173     // connected elements of codim dim
174     int vertices () {return asEngine.vertices();}
175     typename ElementPtrEngine<Bonds,dim,dim>::Engine vertex (int i)
176         { return asEngine().vertex(i);}
177 
178     // hierarchic access to higher level elements, maxlevel = all levels <=maxlevel
179     typename HierarchicIteratorEngine<Bonds,codim,dim>::Engine hbegin (int maxlevel) {
180         return asEngine().hbegin<codim>(level);
181     }
182     typename HierarchicIteratorEngine<Bonds,codim,dim>::Engine hend (int maxlevel) {
183         return asEngine().hend<codim>(level);
184     }
185 
186     // access to father element on next coarser level, same codimension
187     typename ElementPtrEngine<Bonds,codim,dim>::Engine father ()
188         { return asEngine().father();}
189 
190 private:
191     Engine& asEngine() { return static_cast<Engine&>(*this);}
192 };
193 
194 // vertices
195 template<typename engine, int dim, typename Bonds>
196 class Element<engine,dim,dim,Bonds> {
197 public:
198     // NEVER use first template parameter, it is only used for
199     // getting default template parameters in function call
200     typedef typename ElementEngine<Bonds,dim,dim>::Engine Engine; // my engine
201     enum { dimension=dim };      // know your own dimension
202     enum { codimension=dim }; // know your own codimension
203 
204     int level () {return asEngine.level();}
205     int id ()  {return asEngine.id();}
206     int kind() {return asEngine.kind();}
207     Point<dim> position () {return asEngine.position();}
208 
209     // connected elements of codim 0
210     int elements () {return asEngine.elements();}
211     typename ElementPtrEngine<Bonds,0,dim>::Engine element (int i)
212         { return asEngine().element(i);}
213 
```

```
214     // access to father element on next coarser level
215     // this is an element of codimension 0 and local coordinates
216     Point<dim> local () {return asEngine.local();}
217     typename ElementPtrEngine<Bonds,0,dim>::Engine father ()
218         { return asEngine().father();}
219
220 private:
221     Engine& asEngine() { return static_cast<Engine&>(*this);}
222 };
223
224 //*****************************************************************
225 // ElementPtr base class
226
227 template<typename engine, int codim,
228          int dim = engine::dimension, typename Bonds = typename engine::Bonds>
229 class ElementPtr {
230 public:
231     // NEVER use first template parameter, it is only used for
232     // getting default template parameters in function call
233     typedef typename ElementPtrEngine<Bonds,codim,dim>::Engine Engine; // my engine
234     enum { dimension=dim };      // know your own dimension
235     enum { codimension=codim }; // know your own codimension
236
237     // comparison
238     bool operator== (const Engine& i) const { return asEngine().operator==(i); }
239     bool operator!= (const Engine& i) const { return asEngine().operator!=(i); }
240
241     // access to referenced object
242     typename ElementEngine<Bonds,codim,dim>::Engine& operator*() const
243         {return asLeaf().operator*();}
244     typename ElementEngine<Bonds,codim,dim>::Engine* operator->() const
245         {return asLeaf().operator->();}
246
247 private:
248     Engine& asEngine() { return static_cast<Engine&>(*this);}
249 };
250
251
252 //*****************************************************************
253 // LevelIterator base class
254
255 template<typename engine, int codim,
256          int dim = engine::dimension, typename Bonds = typename engine::Bonds>
257 class LevelIterator {
258 public:
259     // NEVER use first template parameter, it is only used for
260     // getting default template parameters in function call
261     typedef typename LevelIteratorEngine<Bonds,codim,dim>::Engine Engine; // my engine
262     enum { dimension=dim };      // know your own dimension
263     enum { codimension=codim }; // know your own codimension
264
265     // navigation
266     Engine operator++()      {return asEngine().operator++();}
267     Engine operator++(int i) {return asEngine().operator++(i);}
268     Engine operator--()      {return asEngine().operator--();}
269     Engine operator--(int i) {return asEngine().operator--(i);}
270
271     // comparison
272     bool operator== (const Engine& i) const { return asEngine().operator==(i); }
273     bool operator!= (const Engine& i) const { return asEngine().operator!=(i); }
274
275     // access to referenced object
276     typename ElementEngine<Bonds,codim,dim>::Engine& operator*() const
```

```
277         {return asLeaf().operator*();}
278     typename ElementEngine<Bonds,codim,dim>::Engine* operator->() const
279         {return asLeaf().operator->();}
280
281 private:
282     Engine& asEngine() { return static_cast<Engine&>(*this);}
283 };
284
285
286 //*******************************************************************
287 // HierarchicIterator base class
288
289 template<typename engine, int codim,
290          int dim = engine::dimension, typename Bonds = typename engine::Bonds>
291 class HierarchicIterator {
292 public:
293     // NEVER use first template parameter, it is only used for
294     // getting default template parameters in function call
295     typedef typename HierarchicIteratorEngine<Bonds,codim,dim>::Engine Engine; // my engine
296     enum { dimension=dim };      // know your own dimension
297     enum { codimension=codim }; // know your own codimension
298
299     // navigation
300     Engine operator++()      {return asEngine().operator++();}
301     Engine operator++(int i) {return asEngine().operator++(i);}
302     Engine operator--()      {return asEngine().operator--();}
303     Engine operator--(int i) {return asEngine().operator--(i);}
304
305     // comparison
306     bool operator== (const Engine& i) const { return asEngine().operator==(i); }
307     bool operator!= (const Engine& i) const { return asEngine().operator!=(i); }
308
309     // access to referenced object
310     typename ElementEngine<Bonds,codim,dim>::Engine& operator*() const
311         {return asLeaf().operator*();}
312     typename ElementEngine<Bonds,codim,dim>::Engine* operator->() const
313         {return asLeaf().operator->();}
314
315 private:
316     Engine& asEngine() { return static_cast<Engine&>(*this);}
317 };
318
319
320
321 //*******************************************************************
322 // Grid base class
323
324 template<typename engine, int dim = engine::dimension,
325          typename Bonds = typename engine::Bonds>
326 class Grid {
327 public:
328     // NEVER use first template parameter, it is only used for
329     // getting default template parameters in function call
330     typedef typename GridEngine<Bonds,dim>::Engine Engine; // my engine
331     enum { dimension=dim };      // know your own dimension
332
333     int maxlevel () {
334         return asEngine().maxlevel();
335     }
336
337     // levelwise access, level = desired level
338     template<int codim>
339     typename LevelIteratorEngine<Bonds,codim,dim>::Engine lbegin (int level) {
```

```
340         return asEngine().lbegin<codim>(level);
341     }
342     template<int codim>
343     typename LevelIteratorEngine<Bonds,codim,dim>::Engine lend (int level) {
344         return asEngine().lend<codim>(level);
345     }
346
347 public:
348     Engine& asEngine() { return static_cast<Engine&>(*this);}
349 };
350
351 #endif
```

### 2.4 The Implementation for a Structured Grid

```
1  #ifndef __SGRID_HH__
2  #define __SGRID_HH__
3
4  #include"grid.hh"
5
6  // special implementation of the grid interface
7  // for structured parallel grids
8
9  //*******************************************************************
10 // We need four classes
11 // template declarations Stroustrup97 p. 343
12 template<int dim> class SGrid;
13 template<int codim, int dim> class SElement;
14 template<int codim, int dim> class SElementPtr;
15 template<int codim, int dim> class SLevelIterator;
16 template<int codim, int dim> class SHierarchicIterator;
17
18 //*******************************************************************
19 // Bonds: Map types to generic names
20 // Here Bonds class is a template depending on dimension
21 template<int dim>
22 struct SGridBonds {
23     enum { dimension=dim };
24 };
25
26 // define bonds for each dimension through specialization
27 template<> struct SGridBonds<1> {
28     enum { dimension=1 };
29     typedef SGrid<1> Grid;
30     typedef SElement<1,1> Vertex;
31     typedef SElement<0,1> Element;
32     typedef SElementPtr<1,1> VertexPtr;
33     typedef SElementPtr<0,1> ElementPtr;
34     typedef SLevelIterator<1,1> VertexLevelIterator;
35     typedef SLevelIterator<0,1> ElementLevelIterator;
36     typedef SHierarchicIterator<1,1> VertexHierarchicIterator;
37     typedef SHierarchicIterator<0,1> ElementHierarchicIterator;
38 };
39
40 template<> struct SGridBonds<2> {
41     enum { dimension=2 };
42     typedef SGrid<1> Grid;
43     typedef SElement<2,2> Vertex;
44     typedef SElement<1,2> Edge;
45     typedef SElement<0,2> Element;
46     typedef SElementPtr<2,2> VertexPtr;
47     typedef SElementPtr<1,2> EdgePtr;
48     typedef SElementPtr<0,2> ElementPtr;
```

```
49      typedef SLevelIterator<2,2> VertexLevelIterator;
50      typedef SLevelIterator<1,2> EdgeLevelIterator;
51      typedef SLevelIterator<0,2> ElementLevelIterator;
52      typedef SHierarchicIterator<2,2> VertexHierarchicIterator;
53      typedef SHierarchicIterator<1,2> EdgeHierarchicIterator;
54      typedef SHierarchicIterator<0,2> ElementHierarchicIterator;
55  };
56
57  template<> struct SGridBonds<3> {
58      enum { dimension=3 };
59      typedef SGrid<1> Grid;
60      typedef SElement<3,3> Vertex;
61      typedef SElement<2,3> Edge;
62      typedef SElement<1,3> Face;
63      typedef SElement<0,3> Element;
64      typedef SElementPtr<3,3> VertexPtr;
65      typedef SElementPtr<2,3> EdgePtr;
66      typedef SElementPtr<1,3> FacePtr;
67      typedef SElementPtr<0,3> ElementPtr;
68      typedef SLevelIterator<3,3> VertexLevelIterator;
69      typedef SLevelIterator<2,3> EdgeLevelIterator;
70      typedef SLevelIterator<1,3> FaceLevelIterator;
71      typedef SLevelIterator<0,3> ElementLevelIterator;
72      typedef SHierarchicIterator<3,3> VertexHierarchicIterator;
73      typedef SHierarchicIterator<2,3> EdgeHierarchicIterator;
74      typedef SHierarchicIterator<1,3> FaceHierarchicIterator;
75      typedef SHierarchicIterator<0,3> ElementHierarchicIterator;
76  };
77
78  //*******************************************************************
79  // here ist the generic ElementPtr class
80  template<int codim, int dim>
81  class SElementPtr :
82      public ElementPtr< SElementPtr<codim,dim>, codim, dim, SGridBonds<dim> >
83  {
84  public:
85      typedef SGridBonds<dim> Bonds; // import all other type names
86      enum { dimension=dim };          // know your own dimension
87      enum { codimension=codim };      // know your own dimension
88
89      // constructor
90      SElementPtr();
91
92      // comparison
93      bool operator== (const SElementPtr<codim,dim>& i) const;
94      bool operator!= (const SElementPtr<codim,dim>& i) const;
95
96      // access to referenced object
97      SElement<codim,dim>& operator*() const;
98      SElement<codim,dim>* operator->() const;
99  private:
100     SElement<codim,dim> virtual_element;
101 };
102
103 //*******************************************************************
104 // The general Element template
105
106 // elements of codim!=0
107 template<int codim, int dim>
108 class SElement : public Element< SElement<codim,dim>, codim, dim, SGridBonds<dim> >
109 {
110 public:
111     int level();            // level of element
```

```
112     int id ();                // unique identification
113     int kind();               // triangle, tetrahedron, see reference topology
114 
115     // hierarchical access to subelements
116     int subelements ();
117     SElementPtr<codim+1,dim> subelement (int i);
118 
119     // connected elements of codim 0
120     int elements ();
121     SElementPtr<0,dim> element (int i);
122 
123     // connected elements of codim dim
124     int vertices ();
125     SElementPtr<dim,dim> vertex (int i);
126 
127     // hierarchic access, maxlevel = all levels <=maxlevel
128     SHierarchicIterator<codim,dim> hbegin (int maxlevel);
129     SHierarchicIterator<codim,dim> hend (int maxlevel);
130 
131     // father element on next coarser level
132     SElementPtr<codim,dim> father ();
133 };
134 
135 // vertices in all dimensions
136 template<int dim>
137 class SElement<dim,dim> : public Element< SElement<dim,dim>, dim, dim, SGridBonds<dim> >
138 {
139 public:
140     int level ();             // level of element
141     int id ();                // unique identification
142     int kind();               // triangle, tetrahedron, etc.
143     Point<dim> position (); // where is the vertex
144 
145     // there are no subelements, this function returns 0!
146 
147     // connected elements of codim 0
148     int elements ();
149     SElementPtr<0,dim> element (int i);
150 
151     // father information
152     Point<dim> local ();
153     SElementPtr<0,dim> father ();
154 };
155 
156 //*****************************************************************
157 // here ist the generic LevelIterator class
158 template<int codim, int dim>
159 class SLevelIterator :
160     public LevelIterator< SLevelIterator<codim,dim>, codim, dim, SGridBonds<dim> >
161 {
162 public:
163     typedef SGridBonds<dim> Bonds; // import all other type names
164     enum { dimension=dim };          // know your own dimension
165     enum { codimension=codim };      // know your own dimension
166 
167     // constructor
168     SLevelIterator();
169 
170     // navigation
171     SLevelIterator<codim,dim> operator ++();
172     SLevelIterator<codim,dim> operator ++(int i);
173     SLevelIterator<codim,dim> operator --();
174     SLevelIterator<codim,dim> operator --(int i);
```

```
175
176     // comparison
177     bool operator== (const SLevelIterator<codim,dim>& i) const;
178     bool operator!= (const SLevelIterator<codim,dim>& i) const;
179
180     // access to referenced object
181     SElement<codim,dim>& operator*() ;
182     SElement<codim,dim>* operator->() ;
183 private:
184     SElement<codim,dim> virtual_element;
185 };
186
187 //*******************************************************************
188 // here ist the generic HierarchicIterator class
189 template<int codim, int dim>
190 class SHierarchicIterator :
191     public HierarchicIterator< SHierarchicIterator<codim,dim>, codim, dim, SGridBonds<dim> >
192 {
193 public:
194     typedef SGridBonds<dim> Bonds; // import all other type names
195     enum { dimension=dim };         // know your own dimension
196     enum { codimension=codim };     // know your own dimension
197
198     // constructor
199     SHierarchicIterator();
200
201     // navigation
202     SHierarchicIterator<codim,dim> operator ++();
203     SHierarchicIterator<codim,dim> operator ++(int i);
204     SHierarchicIterator<codim,dim> operator --();
205     SHierarchicIterator<codim,dim> operator --(int i);
206
207     // comparison
208     bool operator== (const SHierarchicIterator<codim,dim>& i) const;
209     bool operator!= (const SHierarchicIterator<codim,dim>& i) const;
210
211     // access to referenced object
212     SElement<codim,dim>& operator*()  const;
213     SElement<codim,dim>* operator->()  const;
214 };
215
216 //*******************************************************************
217 // here is the generic SGrid class
218 template<int dim>
219 class SGrid  : public Grid< SGrid<dim>, dim, SGridBonds<dim> >
220 {
221 public:
222     typedef SGridBonds<dim> Bonds; // import all other type names
223     enum { dimension=dim };         // know your own dimension
224
225     // constructor
226     // H_: size of domain
227     // N_: coarse grid size, #elements in one direction
228     // L_: number of levels 0,...,L_-1, maxlevel = L_-1
229     SGrid (double H_, int N0_, int L_);
230
231     // levels are 0 ... maxlevel()
232     int maxlevel();
233
234     // levelwise access, level = desired level
235     template<int codim>
236     SLevelIterator<codim,dim> lbegin (int level);
237     template<int codim>
```

```
238       SLevelIterator<codim,dim> lend (int level);
239
240   private:
241       double H;  // length of cube in each direction
242       int N0; // number of elements in coarsest grid in one direction
243       int L;  // number of levels in hierarchic mesh
244   };
245
246
247   //**********************************************************************
248   // general coordinate mapper for n-dimensional grids
249   template<class T, int d>
250   class Tupel {
251   public:
252       Tupel() {}
253       int& operator[] (int i) {return x[i];}
254   private:
255       int x[d];
256   };
257
258   template<int dim>
259   class LexOrder {
260   public:
261       // preprocess ordering
262       void init (Tupel<int,dim>& _N);
263
264       // get total number of tupels
265       int tupels ();
266
267       // compute number from a given tupel
268       int n (Tupel<int,dim>& z);
269
270       // compute tupel from number 0 <= n < tupels()
271       Tupel<int,dim> z (int n);
272
273   private:
274       Tupel<int,dim> N; // number of elements per direction
275       int P[dim+1];     // P[i] = Prod_{i=0}^{i} N[i];
276   };
277
278   template<int dim>
279   class JoinOrder {
280   public:
281       // preprocess ordering
282       void init (Tupel<int,dim>& _N);
283
284       // get total number of elements in all sets
285       int size ();
286
287       // compute number from subset and index
288       int n (int subset, int index);
289
290       // compute subset from number
291       int subset (int n);
292
293       // compute index in subset from number
294       int index (int n);
295
296   private:
297       Tupel<int,dim> N;   // number of elements per direction
298       int offset[dim+1];  // P[i] = Sum_{i=0}^{i} N[i];
299   };
300
```

```
301 template<int dim>
302 class CubeMapper {
303 public:
304     // construct with number of elements (of codim 0) in each direction
305     CubeMapper (Tupel<int,dim>& _N);
306
307     // get number of elements in each codimension
308     int elements (int codim);
309
310     // compute codim from coordinate
311     int codim (Tupel<int,dim>& z);
312
313     // compute number from coordinate 0 <= n < elements(codim(z))
314     // general implementation is O(2^dim)
315     int n (Tupel<int,dim>& z);
316
317     // compute coordinates from number and codimension
318     Tupel<int,dim> z (int i, int codim);
319
320 private:
321     Tupel<int,dim> N; // number of elements per direction
322     int ne[dim+1];    // number of elements per codimension
323     int nb[1<<dim];   // number of elements per binary partition
324     int cb[1<<dim];   // codimension of binary partition
325     LexOrder<dim> lex[1<<dim];      // lex ordering within binary partition
326     JoinOrder<1<<dim> join[dim+1]; // join subsets of codimension
327
328     int power2 (int i) {return 1<<i;}
329     int ones (int b); // count number of bits set in binary rep of b
330     int partition (Tupel<int,dim>& z); // get binary representation of partition
331     Tupel<int,dim> reduce (Tupel<int,dim>& z);
332     Tupel<int,dim> expand (Tupel<int,dim>& r, int b);
333 };
334
335 #include"sgrid.cc"
336
337 #endif
```

### 2.5 To Do

1. Complete first implementation.
2. Interaction with geometry (i. e. where is the boundary), discretization (boundary conditions).
3. Interaction with linear algebra.
4. Implement a real discretization.
5. Discussion of design. What should be provided by the element? I know that there are still some errors in the design
6. The code should compile with g++. There are two possibilities: (1) Wait until g++ is fixed (I wrote to the gcc-bugs list), (2) Change the design (Christian is exploring that).
7. documentation of the results. I think that we should have overview articles like this, exploring general concepts, not only doxygen generated documentation.

## References

[BN94] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++.* Addison-Wesley, 1994.