ETH Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Concept Oriented Design of Numerical Software

C. Lage

Research Report No. 98-07
June 1998

Seminar für Angewandte Mathematik
Eidgenössische Technische Hochschule
CH-8092 Zürich
Switzerland

Concept Oriented Design of Numerical Software[1]

C. Lage

Seminar für Angewandte Mathematik
Eidgenössische Technische Hochschule
CH-8092 Zürich
Switzerland

Research Report No. 98-07 June 1998

Abstract

The continuously growing computing power of modern computers admits to tackle numerical problems of extreme complexity. This complexity carries over to the numerical methods applied to solve the problems. Whereas the mathematical formulation of these methods does not raise any difficulties, their implementation turns out to be the bottleneck in the realization of numerical applications.

In the last years, in order to afford relief, object oriented methods were applied to promote reusable and extensible numerical software, since this kind of flexibility is the key to manage complexity. It became evident that a carefully chosen modularization of the considered methods is a necessary requirement to provide flexible software components.

In this paper we give a brief review of object oriented methods to identify the key issues that support a flexible software design and discuss a modularization technique based on mathematical concepts. Finally, the application of this concept oriented approach to boundary element methods is presented.

[1]Presented at the 14th GAMM-Seminar on *Concepts of Numerical Software*, Christian-Albrechts-Universität Kiel, January 23th to 25th, 1998 (in press in *Notes on Numerical Fluid Mechanics*, Vieweg, 1998).

1 Introduction

It is the concern of Numerical Analysis to develop and analyse algorithms solving mathematical problems. Since the algorithms are intended for the design of software to solve the initial problem, they have to meet several requirements. The problems of stability, for example, affect the design of an algorithm as well as limited resources. Here, one could think of the vast memory requirements of boundary element methods which led to the development of cluster and wavelet algorithms or more general the development of parallel algorithms to address limited computing power. Hence, limited resources result in complex algorithms to overcome the restrictions.

But not only limitations also the continuously growing computing power of modern computers increases the complexity of the algorithms, because now we are able to tackle numerical problems of high complexity which carries over to the algorithms applied to solve the problems.

However, with growing complexity the implementation step gains in significance. Today, the design of numerical software must be considered as an autonomous task in the realization of numerical methods, which is one of the major concerns of Scientific Computing. Unfortunately, it turns out to become a true bottleneck blocking the employment of modern methods in industrial applications.

Indeed, there already exists a huge number of software packages which can be used to tackle extensive problems. Due to a modular structure, these packages may be modified to meet the requirements of an extended problem setting. But in most cases this kind of flexibility exists only in the scope of the package and its initial objective. In general, the combination of modules of different packages to realize a new type of problem is not possible or raises many difficulties such that the result is not satisfying.

But reuse of software components, even if they are designed in a different context than the current one, and extensibility are the fundamental requirements to manage the complexity of modern numerical methods and applications. They also imply an increased reliability of the constructed software, since we refer to components that are already tested. In addition, we are able to provide an executable prototype with limited functionality in early stages of the development process. Hence, experiences with this prototype can be exploited in the software design.

TYPES
Stack[X]
FUNCTIONS
empty: Stack[X] $\longrightarrow$ Boolean
new: $\longrightarrow$ Stack[X]
push: X $\times$ Stack[X] $\longrightarrow$ Stack[X]
pop: Stack[X] $\longrightarrow$ Stack[X]
top: Stack[X] $\longrightarrow$ X
PRECONDITIONS
pre *pop*(s: Stack[X]) = (**not** empty(s))
pre *top*(s: Stack[X]) = (**not** empty(s))
AXIOMS
For all x: X, s: Stack[X]:
empty(*new*())
not *empty*(*push*(x, s))
top(*push*(x,s)) = x
pop(*push*(x,s)) = s

Figure 1: Abstract data type specifying stacks [5].

2 Object Oriented Methods

The key to achieve the aims of reusability and extensibility is as already indicated modularity, which means the construction of software by almost independent or loosely coupled components, so-called modules. Clearly, these modules are intended for reuse whereas the extension of software is given by adding or replacing modules in the software architecture.

With continuously growing demands on the flexibility of the software design the specification of modules has changed. Initially, in the sense of modular programming, modules were considered as a set of correlative data structures and procedures collected in a file. This promotes the principle of information hiding. Nowadays, in the object oriented methodology, a module is specified by an abstract data type. That means modularization is based on units of fine grain.

The expression *abstract data type* stands for a specification of a class of data structures not by an implementation as indicated by the term abstract, but by a list of operations and properties, which are available on the data structures. Let us consider an example shown in Figure 1. A stack is characterized only by means of the operations which observe the *last in first out* policy, e.g., *push a new element*, *pop the top element* or *test whether the stack is empty*. Preconditions could be used to specify whether an operation

is available or not. Since declaring functions does not provide any semantic information associated with the abstract data type, axioms may be added.

In the terminology of object oriented methods the formal specification by means of abstract data types coincides with the definition of classes [1], to say, classes describe or form modules.

Note that with respect to the representation of modules nothing essential has changed. An abstract data type may be expressed with the mechanisms of modular programming languages as well as with the usage of classes in modern object oriented languages. However, the latter will be more convenient. What has changed is the meaning of a module, namely from the grouping of data structures and procedures in order to structure programs to the classification of structures – now, each module serves to produce any number of instances, in object oriented terminology *objects*, of what they stand for.

Modularity is a necessary property to promote reusability and extensibility. However, the plain modularity described so far is not sufficient to achieve a high degree of flexibility. To this end, we must be able to formulate *generic modules*, i.e. modules which serve to capture common properties of related modules and make these available to others. One may think of an interface to a group of modules.

Generic modules form a *generalization* of modules they group whereas the modules themselves are *specializations* of the generic module. It is exactly this property to express hierarchies of modules which distinguishes object oriented methods from traditional approaches.

To illustrate this kind of association we consider two modules, one representing a stack and the other representing a queue (cf. Figure 2). Both structures have in common that we can *put* elements on the stack or in the queue and that we can *get* elements back. The only difference is that a stack observes the *last in first out* whereas the queue provides the *first in first out* strategy. The common operations, put and get, however, specify the generic module *bag*. Users of this module can rely on the fact that they can put something in the bag and that they will get it back. This contract, for instance, can be used by a further module representing the traverse of trees, hence, a kind of iterator, where the following algorithm is used:

```
next() {
  n = b.get()
  for all children c of n
    b.put(c)
  return n
}
```

[1] Most object oriented languages neglect the feature to specify semantics.

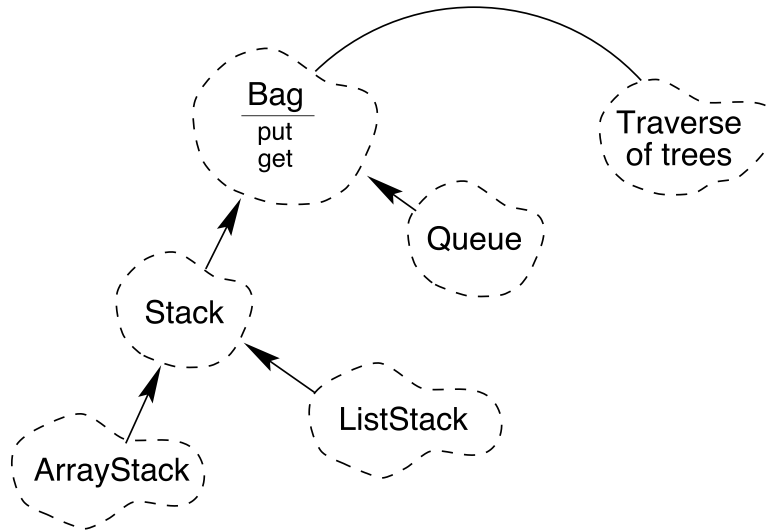

Figure 2: Usage of generic modules.

Under the assumption of a proper initialization of the bag we have to get in each step a node from the bag, put, if available, all children of the node into the bag and return the node. Note that we have only used the operations *put* and *get*. Therefore, we may replace the *bag* by the specializations *stack* or *queue*. The former would result in a *depth first* traverse of the tree, the latter in a *breadth first* traverse. So we modify the type of the traverse without changing the iterator itself or in other words we could easily extend software by attaching specialized modules to generic modules. Similarly, varying implementations could be managed. The implementation of a stack, for example, by means of arrays in order to model a bounded but efficient version of a stack, or by means of lists to obtain an unbounded realization.

In the object oriented methodology generic modules are essentially provided by two mechanisms:

- *type extension* (inheritance, overloading) in connection with *polymorphism* (dynamic binding, dynamic types) and
- *parameterized types* (templates).

3 Concept Oriented Design

As discussed in the previous sections we must organize our software using generic modules specified by abstract data types in order to develop reusable

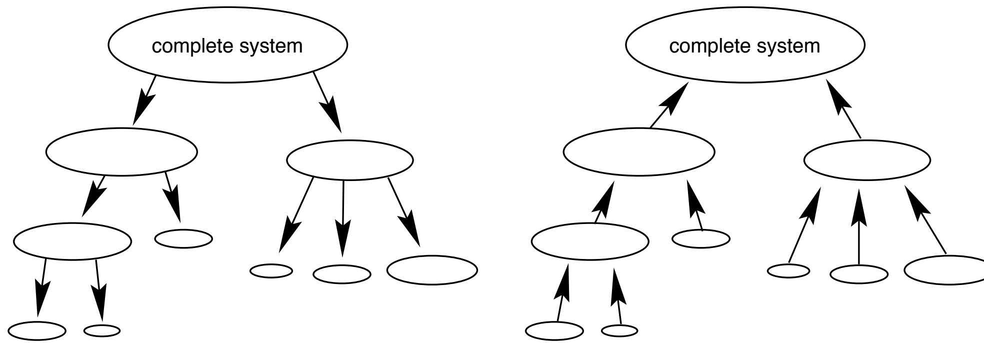

Figure 3: Identifying modules by decomposition and composition.

and extensible software. The tools necessary to follow this rule are provided by object oriented methods. However, the question how to identify or how to construct a suitable modularization of a system arises immediately.

The first steps in applying object oriented methods for the design of numerical software employed the same modules that were already known from the traditional organization. But this approach merely led to a convenient implementation without exhausting the new design method such that no advantage became evident.

In the following we are going to discuss two approaches to identify modules known as

- decomposition (top-down) and
- composition (bottom-up).

The decomposition technique (cf. Figure 3) starts with a module enclosing the complete system. This root module is recursively decomposed into submodules until we reach a level of abstraction that could be easily implemented. The advantage of this technique becomes clear immediately: at the same time with the decomposition of modules we reduce the complexity of the initial problem, therefore, problems with high complexity become manageable. However, since the submodules are typically defined in the scope of their generating modules, the decomposition, in general, does not produce modules that are likely to be reused in another context.

The other way round, starting the design with elementary structures in order to combine these to build abstractions of a higher level, as it is intended by the composition technique, involves the danger to construct modules that cannot be associated to entities of the problem domain. Hence, a small variation of the problem formulation may result in a redesign of several modules

showing that the degree of extensibility is low. In addition, no reduction of complexity is achieved by this approach rendering this technique only applicable for small problem sizes.

Obviously, these two methods could not give a satisfying recipe for identifying modules and it becomes evident why this task is the crucial point in object oriented methods. Nevertheless, since we are interested in the development of numerical software, there is a special situation: the considered numerical methods are already formulated in an abstract way based on hierarchical structured mathematical concepts. This motivates the following approach: represent each concept by a module and combine these modules according to the numerical algorithm to generate an implementation. This defines *concept oriented* modularization. Chances are that the high reusability and extensibility of mathematical concepts carry over to the modules. In other words, we use the mathematical formulation to predict a stable modularization of the complete system.

Unfortunately, we are confronted with the following problem: mathematical concepts are organized in hierarchies, for example, constant functions may be generalized to polynomials and polynomials to functions. Each generalization step results in a reduction of information that may cause a severe loss of efficiency. Consider the concept of a derivative implemented by numerical differentiation, for example. The application of this operator to functions yields a proper realization. On the other hand, if the function is a polynomial or even a constant function this implementation is highly inefficient. The obvious solution to overcome this problem is to preserve or recover the lost type information, in order to select an appropriate implementation. The latter could be related to the polymorphism by virtual functions of object oriented methods. However, a slightly more general mechanism similar a generic dispatch[2] is necessary as we will see in the next section.

4 Example: Petrov-Galerkin Discretization

In order to illustrate the sketched design technique let us consider a Petrov-Galerkin discretization, the foundation of every finite element and boundary element method. Let F and G denote two Hilbert spaces, usually spaces of functions, $A : F \mapsto G$ an operator. To solve the operator equation $Af = g$ numerically, where $g \in G$ denotes a given right hand side, we choose finite dimensional subspaces F_n and G'_n of the space F and the dual space of G, respectively. Then, the Petrov-Galerkin discretization of the problem is given

[2] The programming language CLOS offers this kind of polymorphism.

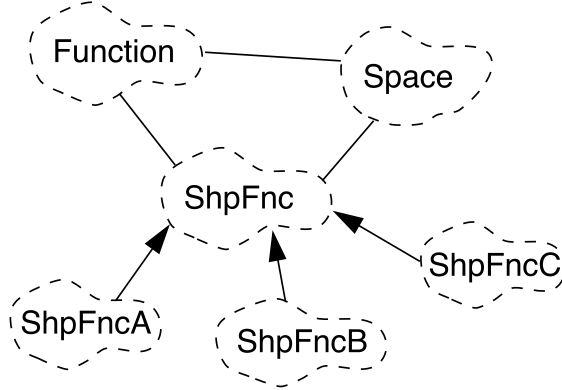

Figure 4: Generic module ShpFnc and associated implementations.

by:

$$\text{find } f_n \in F_n, \text{ such that } (\varphi, Af_n - g) = 0 \text{ for all } \varphi \in G'_n, \tag{1}$$

i.e., such that the projection of the defect vanishes. Here, the projection is determined by the choice of the subspaces. Introducing bases

$$F_n = \text{span}\{\psi_1, \dots, \psi_n\}, \quad G'_n = \text{span}\{\varphi_1, \dots, \varphi_n\} \tag{2}$$

for the subspaces leads to a system of linear equations for the coefficients $\mathbf{f}$ of the discrete solution $f_n = \sum_j (\mathbf{f})_j \, \psi_j$:

$$\mathbf{A}\,\mathbf{f} = \mathbf{g}, \tag{3}$$

where $(\mathbf{A})_{ij} = (\varphi_i, A\psi_j)$ and $(\mathbf{g})_i = (\varphi_i, g)$.

Note that the system matrix may be dense, even if we use basis functions with local support, since the application of A does not necessarily preserve this property.

The major concepts of the discretization scheme are: operators, functions, subspaces and basis. In addition, for the definition of the spaces we have to provide some information about the geometry, in particular, the used mesh or panelization. If we assume that basis functions are represented by shape functions, which are basis functions restricted to a panel, then a central module will be a class ShpFnc specifying a generic module, such that, similarly to the example of the stack, arbitrary implementations of concrete shape functions denoted by ShpFncA, ShpFncB, ShpFncC in Figure 4 may be added.

In the following we are going to discuss the abstraction of the dual forms that must be evaluated in order to assemble the right hand side and system

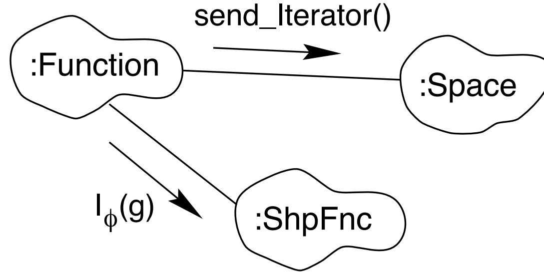

Figure 5: Initialization of discrete functions using shape functions with attached evaluation of $l_\varphi(g) = (\varphi, g)$.

matrix. To be more concrete let us assume that the dual form is given by the L^2 inner product. Then, the elementary operation used to evaluate the right hand side is to integrate the product of right hand side and shape function over an arbitrary panel of the mesh:

$$(\varphi, g) = \int_\triangle g(x)\,\varphi(x)\,dx.$$

At a glance, it is near at hand to attach this integration by means of a virtual function $l_\varphi(g) := (\varphi, g)$ to the class ShpFnc. Virtual functions are a mechanism of object oriented methods to promote generic constructions and ensures in our case that even if the evaluation of the dual form is called for the interface the function call is transfered to the intended concrete subclass which provides the correct implementation (see [7]). In this way we are able to implement the initialization of discrete functions modelled by the class Function without knowledge of concrete shape functions (Figure 5: the initialization calls for an iterator which is able to list all shape functions of the considered space. With this iterator the function $l_\varphi(g)$ is invoked for each shape function. The returned values are processed in order to assemble the vector $\mathbf{g}$ in (3).

Though this construction makes sense, the following problem occurs: assume that not the plain right hand side g, but the image of g with respect to an operator V is to be evaluated in the dual form, i.e. $l_\varphi(g) := (\varphi, Vg)$. This is, for instance, the case for boundary integral equations derived from boundary value problems by means of the direct method. We would have to add new classes of shape functions implementing the new functionality. But this increases the cohesion of the modules. Since spaces are responsible to generate the necessary shape functions to describe the space, we have to add a new space, which is from the mathematical point of view identical with the

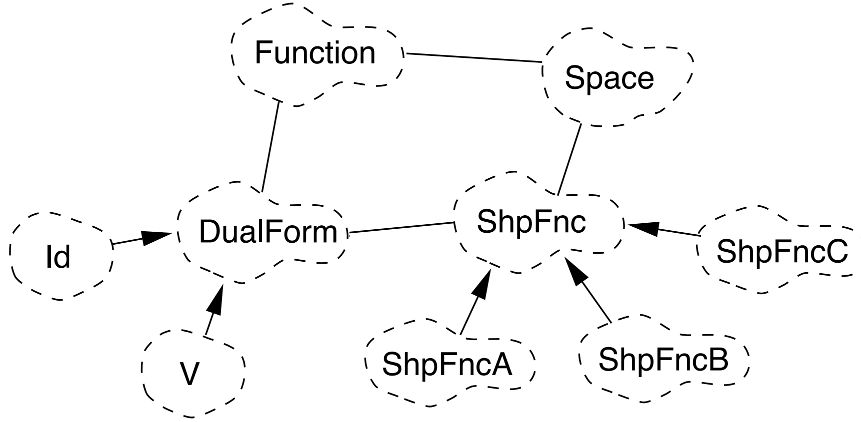

Figure 6: Dual forms as an independent concept.

previous space, only to generate the new shape functions. Thus, the treatment of the right hand side influences assembly and choice of spaces. This association may cause unpredictable difficulties when extending or reusing the design and should be avoided.

The solution is to follow the mathematical structure and realize dual forms as an independent concept. Now, variations in the semantic of dual forms could again easily be modelled by concrete specializations, classes Id and V in Figure 6, without affecting other classes.

But now, another problem arises. Since we divide dual forms from shape functions we have to identify operations supported by shape functions that on the one hand provide enough information such that a dual form can be evaluated, on the other hand are common to all shape functions, in order to form an interface. If we assume, for example, that dual forms can always be evaluated by numerical quadrature, an interface that offers the following information will be possible:

- domain of integration,
- evaluation of mapping functions / Jacobian,
- evaluation of shape functions.

But it turns out that due to its generality such an interface is by far too inefficient, as already indicated. If we implement iso-parametric elements for instance we will not be able to exploit the fact that mapping and shape functions are polynomials, because we are bound to the interface, which can only admit pointwise evaluation of these functions without loosing generality.

So, we are in the above mentioned dilemma that the reduction of information due to a generalization results in a loss of efficiency.

The subsequent solutions to this problem may be conceivable:

First, we may introduce for each case where the loss of efficiency would be too high a separate module. But this is not very convincing, since adding a further module would actuate a redesign of associated classes, similar to the approach by virtual functions already discussed.

Second, use the union of all interfaces that come into question to form a unifying interface, sometimes called a fat interface. This is somewhat more promising, because adding a new interface would only result in a recompilation of associated classes. However, this is still be unacceptable if we plan to design a library. The major drawback of this technique is that the most operations declared by the interface are meaningless in specializations and could only be defined to raise an exception.

Till now, we tried to succeed by preserving the information that might be lost. In the next suggestion we recover the information. For the common interface we choose the intersection of all interfaces and in the implementation of the dual form we add a selection mechanism driven by a type identification of the actual shape function to choose an efficient algorithm. A crude example of such a generic dispatch would be

```
(φ, g) {
   if (typeof(φ) = "ShpFncA") {
      integrate using the interface of ShpFncA
   } else if (typeof(φ) = "ShpFncB") {
      integrate using the interface of ShpFncB
   } else   ···   {
      integrate using the default interface of ShpFnc
   }
}
```

Type identification (*typeof()*) is usually provided by the object oriented programming language chosen or can be easily implemented by means of virtual functions.

This construction supports the desired and necessary flexibility: we could attach any kind of shape functions and dual forms without changing existing classes. Note that we do not by-pass the principle of information hiding by identifying the type, because we still communicate with shape functions using an interface which, however, is more appropriate. Another, more advanced

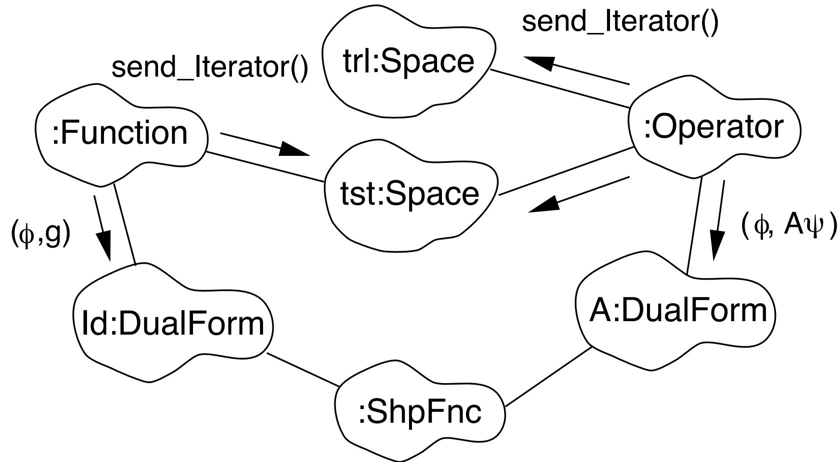

Figure 7: Evaluation of system matrix and right hand side.

implementation of a generic dispatch consists in using the type information as index to retrieve the appropriate function call from a table.

It is straight forward to apply the approach of recovering information to the dual forms of the left hand side to assemble the system matrix. The only difference is that now the selection depends on the types of two shape functions φ and ψ instead of a single one (see Figure 7). In this case an implementation using nested virtual functions (double dispatching) would not be extensible at all (cf. [8, 13.8]).

Summing up, it may be said that the concept oriented design, which offers the possibility to produce a stable, extensible and reusable modularization of numerical problems, needs an extended understanding of polymorphism. Typically, polymorphism is only driven by the declaring type, e.g., a function call is passed from a generic module to the intended specialized module. Now, in a second stage, we must be able to declare generic functions in the scope of modules, i.e. functions whose parameter list contain generic modules. Calling a generic function invokes a specialized function. The selection depends on the actual instance of the generic modules in the parameter list, where we are not restricted to a single module or modules, which are related in a special way to the module declaring the functions.

5 A Framework for Boundary Element Methods

Using the discussed design technique we have develop a C++ class library for boundary element methods called *Concepts*.

The integral equations we are interested in are integral equations of Fredholm type

$$Af = g \text{ on } \Gamma, \quad (Af)(x) := \int_\Gamma k(x,y) f(y)\, dy$$

that originate from the reformulation of a boundary value problem

$$Lu = 0 \text{ in } \Omega \subset \mathbb{R}^d, \quad Bu = \varphi \text{ on } \Gamma := \partial\Omega$$

by means of the integral equation method, described for example in [1]. The transformation is by no means unique. For Laplace's equation with Dirichlet boundary conditions

$$-\Delta u = 0 \text{ in } \Omega \subset \mathbb{R}^3, \quad u = \varphi_D \text{ on } \Gamma := \partial\Omega$$

one could, for instance, formulate the following three boundary integral equations:

ansatz in Ω	equation on Γ
$u = Vf$	$Vf = \varphi_D$
$u = Kf$	$\left(\frac{1}{2}I - K\right) f = -\varphi_D$
$u = V\varphi_N - K\varphi_D$	$V\varphi_N = \left(\frac{1}{2}I + K\right)\varphi_D$

where

$$(Vf)(x) := \frac{1}{4\pi} \int_\Gamma \frac{1}{|y-x|} f(y)\, dy, \tag{4}$$

$$(Kf)(x) := \frac{1}{4\pi} \int_\Gamma \frac{\langle n(y), y-x \rangle}{|y-x|^3} f(y)\, dy. \tag{5}$$

The first and the second formulations are derived using a single and double layer ansatz, respectively, yielding equations for the density f. Applying Green's representation formula leads to the third equation.

In general, the kernel functions are singular for x equal to y and are smooth with growing distance of x and y.

In boundary element methods the discretization of the integral equation is given by Petrov-Galerkin schemes where two choices of subspaces are of major

interest. Choosing Dirac functionals for the test space yields the collocation method, where the dual form is simply a pointwise evaluation in collocation points ξ_i:

$$(\mathbf{g})_i = g(\xi_i), \quad (\mathbf{A})_{ij} = \int_\Gamma k(\xi_i, y)\psi_j(y)\,dy.$$

Choosing the L^2 inner product for the dual form describes the Galerkin method. Hence, in the three dimensional case, we have to evaluate four dimensional singular integrals to assemble the system matrix:

$$(\mathbf{g})_i = \int_\Gamma \varphi_i(y)g(x)\,dy, \quad (\mathbf{A})_{ij} = \int_\Gamma \varphi_i(y)\int_\Gamma k(x,y)\psi_j(y)\,dy\,dx.$$

Since the foundation of the boundary element method is the Petrov-Galerkin scheme, we can make use of the design presented in the previous section. For the abstraction of the geometry, that is the boundary of the domain, we use two stages. The first represents a splitting of the boundary in elementary parts, so-called patches. It represents the physical view of geometry. The second stage, the numerical view, provides an approximation of the boundary, for example, by a polyeder. The panels that could be used in the approximation, like triangles, quadrilaterals, curved triangles, are grouped by the class Panel (Figure 8). Spaces make use of the panels, offered by a panelization via iterators, to generate basis functions which are represented by elements. An element denotes the collection of all shape functions associated to one panel.

Let us have a closer look at the association shared by the class Space and Element (Figure 9). By specialization of the class Space we could attach concrete spaces to the model. During their initialization we select concrete elements according to the semantic of the considered space and establish an index that relates shape functions to basis functions. Functions of the spaces are represented by their vector of coefficients with respect to the chosen basis.

We have already discussed the interplay of dual forms and elements. Dual forms serve to provide the necessary informations to initialize functions and operators.

Operators are characterized by their property to map functions. This attribute is represented by the class Operator (Figure 9). A matrix with matrix vector product, for example, could serve as an implementation of an operator. This representation is given by the class OP_AllPurpose. The name indicates that we could use this class for all discrete operators even for the sparse mass matrix if necessary. But for the latter the class Identity

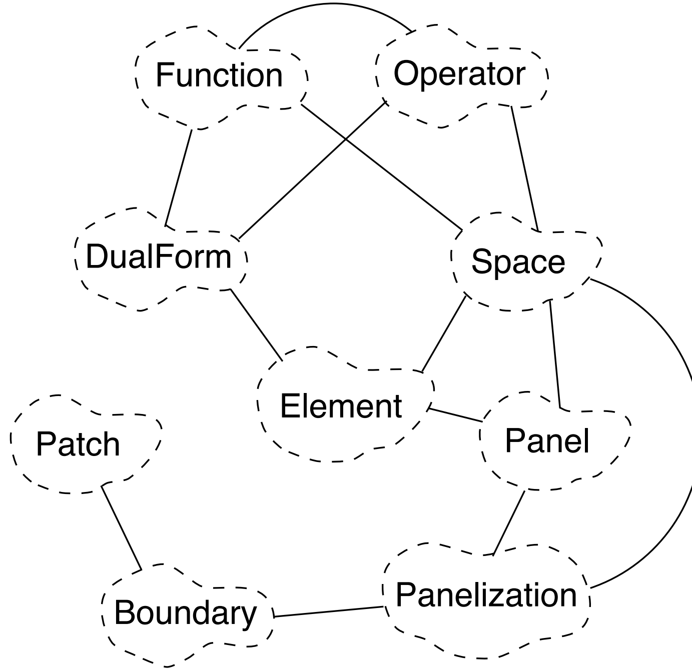

Figure 8: Top level structure of *Concepts*.

was added taking the sparse structure and the simplified initialization into account. Further operators are linear combinations and solvers, since these represent nothing else but the inverse of an operator. The shown iterative solver can be initialized with all classes sharing the interface of an operator, since they only need the mapping property.

With these classes the implementation for solving the boundary integral equation

$$\left(\frac{1}{2}I - K\right) f \;=\; g \quad \text{on } \Gamma \tag{6}$$

could be formulated as follows:

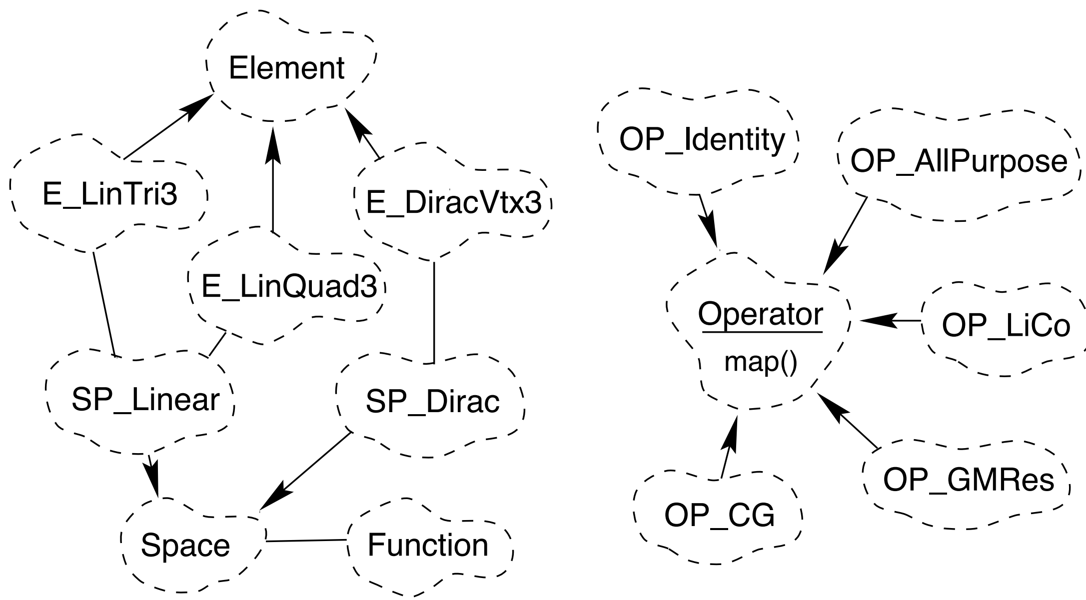

Figure 9: Space–element association and operators.

```
int main() {
   Boundary          gamma("cube");
   Panelization      pnl(gamma, 2);

   SP_Linear         trlspc(pnl);
   SP_Dirac          tstspc(pnl);

   DF_LaplaceDlp     dlp(2, 2, 0.25);
   DF_Identity       id;

   OP_AllPurpose     K(tstspc, trlspc, dlp);
   OP_Identity       I(tstspc, trlspc, id);

   OP_LiCo           A(0.5, I, -1.0, K);
   OP_GMRes          A_inverse(A, 1e-8, 100);

   Function          g(tstspc, id, "(x * y * z)");
   Function          f(trlspc);

   A_inverse(g, f);
}
```

After the generation of physical and numerical informations of the geometry

represented by the objects gamma and pnl, respectively, the trial (trlspc) and test space (tstspc) are chosen. Here, we apply piecewise linear functions and Dirac delta functions concentrated in the vertices, i.e. a collocation method is used for the discretization of (6).[3] In the next two lines the dual form induced by the double-layer potential (cf. (5)) as well as the dual form associated with the identity are declared and defined. In addition, parameters concerning the quadrature rules, e.g., the number of Gaussian nodes, are given.

In the example, the integral operator is described by a linear combination of the identity and the double-layer potential. Again, this representation is carried over into the implementation by defining the object A of the class OP_LiCo. This class is, as already mentioned, responsible for evaluating the linear combination of two operators, in our example the linear combination of the discrete identity and the discrete double-layer potential represented by I and K, respectively.

The integral operator A in combination with further parameters, e.g., a stopping criterion, is used to initialize the solver denoted in the listing by A_inverse. In the last line A_inverse is applied to the given right hand side g to solve equation (6).

It should be mentioned that the object oriented methodology, in particular polymorphism, does not necessarily lead to inefficient implementations. For the presented approach the same efficiency compared to a traditional implementation which offered only a restricted flexibility was reached.

Finally, in order to stress the flexibility of the modularization we want to discuss two extensions. The drawback of the boundary element method in contrast to finite elements is the dense system matrix. Though there is a reduction of the spatial dimension the complexity to assemble the system matrix for boundary elements in the three dimensional case exceeds the complexity of this task in a corresponding finite element discretization of equal mesh width. Due to the special kernel properties, namely the smoothness far from the singularity, it is possible to construct sufficient approximations of the system matrix in $O(N(\log N)^t)$ instead of $O(N^2)$, N the number of unknowns, operations. The same is true for the memory consumption.

One method, the panel clustering method, was proposed by Hackbusch and Novak [2]. It is based on the replacement of the kernel by an approximation in regions where the kernel is smooth, so-called clusters, in order to obtain a factorization of the xy-dependence:

$$k(x,y) \sim k_m(x,y) := \sum_{(\mu,\nu)\in I_m} \kappa_{(\mu,\nu)}(x_0,y_0)\, X_\mu(x;x_0)\, Y_\nu(y;y_0) \tag{7}$$

[3]Setting tstspc = trlspc instead of choosing Dirac delta functions implements Galerkin's method.

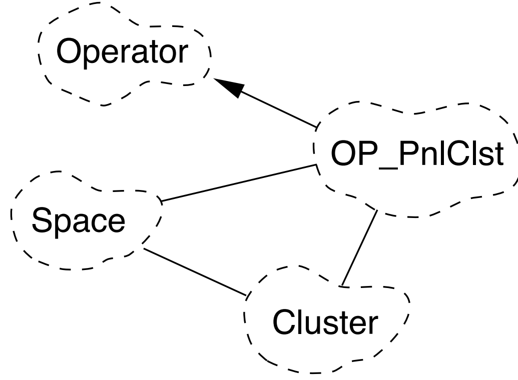

Figure 10: Embedding of the panel clustering method.

for all x, y satisfying

$$|y - y_0| + |x - x_0| \leq \eta\, |y_0 - x_0|.$$

Approximations of this kind could be derived by expansion or interpolation of the kernel. The advantage of the latter is that we only need kernel evaluations to assemble the approximation [3]. Hence, this approach is nearly independent of the kernel.

The approximation of the kernel leads to an approximative decomposition of the system matrix into a sparse matrix $\mathbf{N}$ and a finite sum of rank-M modifications where $M << N$:

$$\mathbf{A} \sim \mathbf{N} + \sum_{(\sigma,\tau)\in\mathcal{F}} \mathbf{X}_\sigma \mathbf{F}_{\sigma\tau} \mathbf{Y}_\tau. \tag{8}$$

Whenever a matrix vector product is necessary we have to evaluate the composed operation

$$\mathbf{Af} \sim \mathbf{Nf} + \sum_{(\sigma,\tau)\in\mathcal{F}} \mathbf{X}_\sigma (\mathbf{F}_{\sigma\tau} (\mathbf{Y}_\tau \mathbf{f})), \tag{9}$$

in order to employ fast methods available for the computation of matrix vector products with rank-M matrices. By evaluating (8) we would obtain a dense matrix and nothing is gained.

The embedding of the panel clustering did not cause any changes of the existing framework. A new implementation of the interface Operator using the panel clustering algorithm and the construction of clusters based on the information provided by the class Space is added (Figure 10). To apply the panel clustering method in the previous example we only have to exchange the OP_AllPurpose operator with the OP_PnlClst operator.

The second method to generate a sparse approximation of the system matrix uses expansions of the kernel by means of wavelets. Again, due to the smoothness of the kernel in the far field regions expansion coefficients corresponding to higher frequencies decrease rapidly. Since these coefficients determine the entries of the system matrix most of them could be neglected. It turns out that such a truncation of sufficient small entries yields a sparse approximation that preserves the convergence rates of the initial method. Under certain assumptions a reduction to $O(N)$ entries is possible (see Schneider [6]). In our library we use discontinuous multi-wavelets that were analysed by von Petersdorff and Schwab [9]; computational aspects are presented in [4]. With this implementation boundary integral equations with up to one million unknowns were solved.

Again, to realize wavelet methods we could extend the given framework by using only specializations of the existing interfaces. A new wavelet element must be made available, a space to setup these elements, dual forms to support special quadrature schemes that become necessary for wavelets and a new operator to realize the truncation and suitable storage schemes. In contrast to panel clustering, most of the major class hierarchies must be extended. But all extensions are caused by the mathematical method itself and not by an improper modularization. In addition, it could be observed that the chosen modularization promotes an incremental design of the concepts. For example, we could check the new space and the new dual form with the existing operator class OP_AllPurpose. In this way one can localize errors more easily.

References

[1] W. Hackbusch. *Integral Equations, Theory and Numerical Treatment*, volume 120 of *ISNM*. Birkhäuser, Basel, 1995.

[2] W. Hackbusch and Z.P. Nowak. On the Fast Matrix Multiplication in the Boundary Element Method by Panel Clustering. *Numerische Mathematik*, 54(4):463–491, 1989.

[3] Ch. Lage. Fast evaluation of singular kernel functions by cluster methods. Technical report, Seminar für Angewandte Mathematik, ETH Zürich, CH-8092 Zürich, 1998. In preparation.

[4] Ch. Lage and Ch. Schwab. Wavelet Galerkin algorithms for boundary integral equations. Technical Report 97-15, Seminar für Angewandte Mathematik, ETH Zürich, CH-8092 Zürich, 1997. To appear in SIAM J. Sci. Comput.

[5] B. Meyer. *Object-Oriented Software Construction.* Series in Computer Science. Prentice-Hall, New York, 1988.

[6] R. Schneider. *Multiskalen- und Wavelet-Matrixkompression: Analysisbasierte Methoden zur effizienten Lösung großer vollbesetzter Gleichungssysteme.* Advances in Numerical Mathematics. Teubner Verlag, Stuttgart, 1998.

[7] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, second edition, 1991.

[8] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, Reading, Massachusetts, 1994.

[9] T. von Petersdorff and Ch. Schwab. Fully discrete multiscale Galerkin BEM. In Dahmen W., Kurdila P., and Oswald P., editors, *Multiresolution Analysis and Partial Differential Equations*, Wavelet Analysis and its Applications. Academic Press, 1997.

Research Reports

No.	Authors	Title
98-07	C. Lage	Concept Oriented Design of Numerical Software
98-06	N.P. Hancke, J.M. Melenk, C. Schwab	A Spectral Galerkin Method for Hydrodynamic Stability Problems
98-05	J. Waldvogel	Long-Term Evolution of Coorbital Motion
98-04	R. Sperb	An alternative to Ewald sums, Part 2: The Coulomb potential in a periodic system
98-03	R. Sperb	The Coulomb energy for dense periodic systems
98-02	J.M. Melenk	On n-widths for Elliptic Problems
98-01	M. Feistauer, C. Schwab	Coupling of an Interior Navier–Stokes Problem with an Exterior Oseen Problem
97-20	R.L. Actis, B.A. Szabo, C. Schwab	Hierarchic Models for Laminated Plates and Shells
97-19	C. Schwab, M. Suri	Mixed *hp* Finite Element Methods for Stokes and Non-Newtonian Flow
97-18	K. Gerdes, D. Schötzau	*hp* FEM for incompressible fluid flow - stable and stabilized
97-17	L. Demkowicz, K. Gerdes, C. Schwab, A. Bajer, T. Walsh	HP90: A general & flexible Fortran 90 *hp*-FE code
97-16	R. Jeltsch, P. Klingenstein	Error Estimators for the Position of Discontinuities in Hyperbolic Conservation Laws with Source Terms which are solved using Operator Splitting
97-15	C. Lage, C. Schwab	Wavelet Galerkin Algorithms for Boundary Integral Equations
97-14	D. Schötzau, C. Schwab, R. Stenberg	Mixed *hp* - FEM on anisotropic meshes II: Hanging nodes and tensor products of boundary layer meshes
97-13	J. Maurer	The Method of Transport for mixed hyperbolic - parabolic systems
97-12	M. Fey, R. Jeltsch, J. Maurer, A.-T. Morel	The method of transport for nonlinear systems of hyperbolic conservation laws in several space dimensions
97-11	K. Gerdes	A summary of infinite element formulations for exterior Helmholtz problems
97-10	R. Jeltsch, R.A. Renaut, J.H. Smit	An Accuracy Barrier for Stable Three-Time-Level Difference Schemes for Hyperbolic Equations
97-09	K. Gerdes, A.M. Matache, C. Schwab	Analysis of membrane locking in *hp* FEM for a cylindrical shell