Chapter 5

Basic tools

A finite element program has several common procedures that can be implemented as basic tools to be used by other part of program. Integrating, calculating shape functions and other geometrical parameters, and linear solvers are some examples of these procedures. Basic tools are defined to implement these common tasks. Implementing basic tools reduces the implementation task by removing the duplicated procedure in program and also increases the reusability and compatibility between different parts of code via their uniform interface.

While many of this tools are used in the most inner part of the code their performance has a great importance. For example integration tools are called inside the `Elements` in time of build and any overhead in their performance will cause a great overhead in program executing time. From other point of view, a large proportion of executing time, is consumed by these tools. A good example is the linear solver which take a large amount of execution time just by itself. All these comments shows the importance of performance tuning in these tools.

Another important requirement for these tools is efficiency in memory. Again in a usual finite element program, a large proportion of used memory is consumed by these tools. This come from two facts, first, a large amount of these tools may be necessary to handle a finite element problem and second, they use a large amount of memory for their operations. For example many geometries must be created in order to model a real problem. So their efficiency in using memory is crucial in order to control the total memory used by program.

The last but not least feature is reusability of these tools. While these tools must be used by different part of the code, their generality and flexibility plays an important role in their successful usage.

In this section the design and implementation of different tools in Kratos will be explained.

5.1 Integration tools

Integrating some function over the domain or boundary is one of the fundamental operations in the FEM. `Elements`, `Conditions` and sometimes other parts of the code have to perform integration in an efficient way. All this makes necessary the designing and implementing an efficient integration tool which can handle different methods of integration with less overhead as possible. Before starting with designing integration tools let take a brief look to numerical integration methods used in finite element programs.

5.1.1 Numerical Integration Methods

It's clear that integrating a function analytically in many cases contains difficulties and in general is not always possible. This made numerical integration, also called quadrature, to be started in 18th and 19th centuries. However use of numerical integration for real problems was postponed to the time of development of computers.

A typical approach to approximate an integral of a function is to evaluate it in different points, apply specific weight to them and sum the weighted values to obtain the result.

$$\int_a^b f(x)\,d(x) \approx \sum_{i=1}^{n} w_i f(x_i) \tag{5.1}$$

Many classical method assume that sample points are in the same distance h all the time.

$$x_i = x_0 + ih \tag{5.2}$$

They use different number of sample points and apply different weighting coefficient to obtain the result. Here there are some examples of methods in this category:

Classical Formulas

A very simple case is trapezoidal rule. It take to sample point and evaluate their corresponding points on the function then connect them with a line and simply calculate the area of trapezoidal obtained below this line. In the other word, it use the average of these two values to calculate the integral:

$$\int_a^b f(x)\,d(x) = \frac{h}{2}f(a) + \frac{h}{2}f(b) + O\left(h^3 f''\right) \tag{5.3}$$

It can be easily seen that this integration is exact for linear function and for other functions while second derivative is unknown to us just approximate the result

$$\int_a^b f(x)\,d(x) \approx \frac{h}{2}f(a) + \frac{h}{2}f(b) \tag{5.4}$$

According to equation 5.1 we can define sample points and weighted for this method:

$$x_1 = a, x_2 = b \tag{5.5a}$$

$$w_1 = w_2 = \frac{h}{2} \tag{5.5b}$$

Famous Simpson's rules are also in this category. They use more sample point to produce higher order approximation. Here is the first one:

$$\int_a^b f(x)\,d(x) = \frac{h}{3}f(a) + \frac{4h}{3}f(\frac{a+b}{2}) + \frac{h}{3}f(b) + O\left(h^5 f^{(4)}\right) \tag{5.6}$$

which is exact for a polynomial up to degree 2. While it approximate other functions better than trapezoidal rule:

$$\int_a^b f(x)\,d(x) \approx \frac{h}{3}f(a) + \frac{4h}{3}f(\frac{a+b}{2}) + \frac{h}{3}f(b) \tag{5.7}$$

Also we can introduce this in the global form using:

$$x_1 = a, x_2 = \frac{a+b}{2}, x_3 = b \tag{5.8a}$$

$$w_1 = w_3 = \frac{h}{3}, w_2 = \frac{4h}{3} \tag{5.8b}$$

There are also extension to this methods using the same contest which approximate with higher order using more sample points. In any case to give exact integral of a polynomial of order n this methods use $2n+1$ sample points. This make them very expensive while there are other ways to achieve the same result with just n sample points, which will be described as follow.

Gaussian Quadrature

As mentioned before in previous methods, sample points assumed to be located in the same distance. If we take this restriction and chose their position in some other manner we can duplicate the order of approximation. This is the base idea of Gaussian quadrature formulas.

But how this magical method works? Here is a quick review over it. First, lets define a weighted scalar product of two functions f over g as

$$\langle f \mid g\rangle = \int_a^b W(x)f(x)\,g(x)dx \tag{5.9}$$

A set of polynomials can be funded which include exactly one polynomial $p_j(x)$ of order j, for each $j = 1, 2, 3, ...$ which are also mutually orthogonal over a given weight function $W(x)$.

Let extend the equation 5.1 to a more general case which is:

$$\int_a^b W(x)f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i) \tag{5.10}$$

Here $W(x)$ is added as a known weighting function applied to our integrand. This extension is to use a powerful feature of Gaussian quadrature where we can make an exact integral over a polynomial times some known weighting function $W(x)$. This weighting function help us to discomposing some integrable function to a polynomial and a complex but integrable part to make an exact integral. Also it can be chosen to remove singularities which are integrable. By the way we can assume equation 5.1 a special case of 5.10 with $W(x) \equiv 1$. Choosing this weighting function results so called *Gauss-legendre integration.*

Now it's time to apply the fundamental theorem of Gaussian quadratures, the abscissas of the N-point Gaussian quadrature formulas with weighting function $W(x)$ in the interval (a, b) are precisely the roots of the orthogonal polynomial $p_n(x)$ for the same interval and weighting function.

For giving a practical way to find Gaussian weight and abscissas, we use a set of orthogonal polynomials defined as

$$p_0(x) \equiv 1 \tag{5.11a}$$

$$p_1(x) = [x - \frac{\langle xp_0|p_0\rangle}{\langle p_0|p_0\rangle}]p_0(x) \tag{5.11b}$$

$$p_{i+1}(x) = [x - \frac{\langle xp_i|p_i\rangle}{\langle p_i|p_i\rangle}]p_i(x) - \frac{\langle p_i|p_i\rangle}{\langle p_{i-1}|p_{i-1}\rangle}p_{i-1}(x) \tag{5.11c}$$

It can be shown that each polynomials $p_j(x)$ has exactly j distinct roots and also there is exactly one root of $p_{j-1}(x)$ between each adjacent pair of them. This property comes very useful in the time of finding roots. A root finding scheme can be applied starting from $p_1(x)$ and continuing to higher orders using interval of previous roots to find all of them. Finally equation 5.10 used to make a system of equations for finding weights w_i. Considering that equation 5.10 must gives the correct answer for the integral of the first $N-1$ polynomials

$$a_{ij}w_j = b_i \tag{5.12a}$$

$$a_{ij} = p_{i-1}(x_j) \tag{5.12b}$$

$$b_i = \int_a^b W(x)p_{i-1}(x)\,dx \tag{5.12c}$$

$$i = 1, ..., N, j = 1, ..., N \tag{5.12d}$$

It can be shown that using the same weights w_i the quadrature is exact for all polynomials of order up to $2N-1$. Also in equation 5.12a it's easy to verify that b_i for $i = 2, ..., N$ is equal to zero considering the orthogonal nature of p_i's.

There is also another, and more practical, way to find the weights w_i, using another sequence of polynomials $\varphi_i(x)$

$$\varphi_0(x) \equiv 0 \tag{5.13a}$$

$$\varphi_1(x) = p_1' \int_a^b W(x)dx \tag{5.13b}$$

$$\varphi_{i+1}(x) = [x - \frac{\langle xp_i|p_i\rangle}{\langle p_i|p_i\rangle}]\varphi_i(x) - \frac{\langle p_i|p_i\rangle}{\langle p_{i-1}|p_{i-1}\rangle}\varphi_{i-1}(x) \tag{5.13c}$$

Where p_1' is the derivative of $p_1(x)$ with respect to x. Using these polynomials calculating the weights is straightforward

$$w_i = \frac{\varphi_N(x_i)}{p_N'(x_i)}, i = 1, ..., N \tag{5.14}$$

Also there is a formula just for Gauss-Legendre case

$$w_i = \frac{2}{(1 - x_i^2)[p'_N(x_i)]^2} \tag{5.15}$$

Multidimensional Integrals

Passing from one dimensional integrals to multidimensional one consist of multiplying number of sample points and also complexity of applying boundaries. In this part just an easy case with simple boundary condition will be discussed. In FEM usually we map our geometry functions to some local coordinates and then reducing the integral to lower dimensionality using simplicity of boundaries in this local coordinate and also considering smoothness of our function. Considering a three dimensional case

$$F \equiv \int_{x_1}^{x_2} \int_{y_1(x)}^{y_2(x)} \int_{z_1(x,y)}^{z_2(x,y)} f(x, y, z) dxdydz \tag{5.16}$$

It can be reduced in 2 dimensional form

$$F = \int_{x_1}^{x_2} \int_{y_1(x)}^{y_2(x)} G(x, y) dxdy \tag{5.17a}$$

$$G \equiv \int_{z_1(x,y)}^{z_2(x,y)} f(x, y, z) dz \tag{5.17b}$$

And finally to one dimensional form

$$F = \int_{x_1}^{x_2} H(x) dx \tag{5.18a}$$

$$H(x) \equiv \int_{y_1(x)}^{y_2(x)} G(x, y) dy \tag{5.18b}$$

It can be seen that for implementing this we need to calculate subdimensional integrals in each sample point.

Finally, using Gaussian quadrature in this approach, results

$$F = \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k=1}^{N} w_i w_j w_k f(x_i, y_j, z_k) \tag{5.19}$$

5.1.2 Existing Approaches

There are several ways to implement integration methods in a program. They can be implemented using a very rigid structure or very flexible and dynamic one. In this part different approaches for implementing integration methods are discussed.

The first and most static and also rigid one is just to introduce manually the integration coordinates and weights in the integrand functions. This approach is used in finite element programs

to create an array of shape functions values in all integration points or directly into the weak form to integrate it over element. The advantage is integration will be extremely fast especially for low order simple elements, while all the values are known at compile time and all the loops can be eliminated. The drawback is that maintaining these structure for higher order elements is difficult and the code is not reusable for other formulations or elements. Here is an example:

```
double f(double x, double y) {
  // function body ...
}

double integral_of_f() {
  const double x1 = -sqrt(1.00 / 3.00)
  const double x2 =  sqrt(1.00 / 3.00)
  // w = 1.00

  return f(x1,x1) + f(x1,x2) +
         f(x2,x1) + f(x2,x2);
}
```

The previous approach can be enhanced by encapsulating the 1 dimensional sample points sets and their weights in a class and creating any n dimensional instance of it just in the dimension loops over integrand. In this manner the integration points can be reused in other parts of program but program still is depended to a set of outer product of 1 dimensional sample points. In other words having a specific n dimensional points is not supported in this manner.

```
array<double, 2>
  Gauss2 = {-sqrt(1.00 / 3.00),
             sqrt(1.00 / 3.00)};


double f(Point& x) {
  // function body ...
}

double integral_of_f() {
  double result = 0;
  const int size = Gauss2.size();
  // w = 1.00

  for(int i = 0 ; i < size ; i++)
    for(int j = 0 ; j < size ; j++)
    {
      Point g(Gauss2[i], Gauss2[j]);
      result += f(g);
    }

  return result;
}
```

Another alternative is to encapsulate an array of quadrature points and their weights in a quadrature class and implement an instance of this class for each set of integration points. This design is also common and relatively more reusable than previous one. The performance is highly depended on how these quadratures object are implemented and used but can be optimized as much

as previous approach. Its weakness is hand coding and also debugging it especially when extending it from 1 dimension to 2 and 3. In this cases for any set of gaussian integration sample points different quadrature sets for 1 and 2 and 3 dimensional integration space must be implemented and tested separately.

```
class Gauss2D2 : public array<Point<2>, 4> { public:
  Gauss2D2()
  {
    // Initializing array with
    // integration points
  }
};

double f(Point<2>& x) {
  // function body ...
}

double integral_of_f() {
  double result = 0;
  const int size = Gauss2D2::size();
  Gauss2D2 gauss_points;
  // w = 1.00

  for(int i = 0 ; i < size ; i++)
    result += f(gauss_points[i])
  return result;
}
```

A good extension to previous design is to automating the construction of 2 and 3 dimensional integration points sets from their 1 dimensional set. This extension make us a new alternative with more compact implementation which is easier to extend and also test. In Deal II [21] this approach is used to implement quadrature classes. The quadrature base class is in charge of expanding 1 dimensional set to its dimension (given by template parameter) and store them while providing the interface for them. Extensibility is the good point of this structure. Any new set of integration points can be added just by deriving it from general base class and any n dimensional set created just by providing abscissas and weights and without changing the library.

5.1.3 Kratos Quadrature Library Design

Kratos quadrature's structure is very simple and straight forward. It consist of two classes `IntegrationPoint` and `Quadrature` with a set of classes representing array of sample points which referred as `IntegrationPoints`.

The library is divided in two parts, one is the library by itself which is not for modifying in term of extension and other is the extension which is the port for adding new methods in order to extend the library. This separation allows to encapsulate all difficulties in the library part and leave the extending part as simple as possible. This was one of the reasons not to use a CRT pattern in its design.

Each abscise and its corresponding weighting encapsulates in `IntegrationPoint` class. In more detail, `IntegrationPoint` derived from `Point` class which makes it possible to be passed as a point to any geometrical function. Also `Point` by itself is an interface and derived from `array_1d`

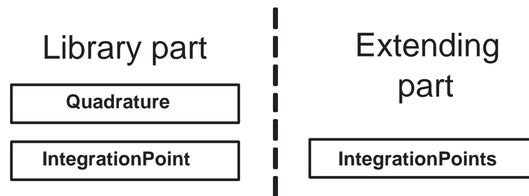

Figure 5.1: Quadrature overall scheme

which physically contains the coordinates value. All these make an `IntegrationPoint` usable in places where an array of coordinates or points needed. In Kratos also having a corresponding weight completes the encapsulation level in terms of storing, passing and using it to integrate a function. `IntegrationPoint` is a template with 3 parameters:

- `TDimension` is an unsigned integer which represents the dimension of this integration point as a point without weight as an extra dimension.
- `TDataType` is the coordinate type of integration point which is a `double` by default.
- `TWeightType` is the type of integration weight stored at an integration point and also is a double by default.

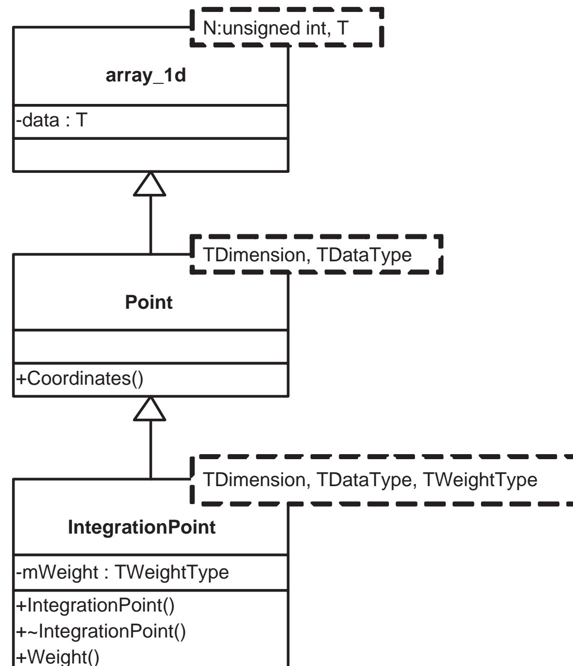

Figure 5.2: `IntegrationPoint` class

`IntegrationPoints` is just an specific set of integration points corresponding to some certain

integration algorithms. It is also the extending point of the library. Any class containing a set of user specified integration points can be used as integration points if confirming these conditions:

- Having a `Dimension` attribute known at compilation time.
- An static `size()` method must be provided and size must be known at compilation time.
- 0 base indexing is also required.

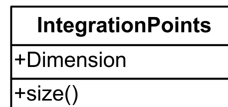

Figure 5.3: IntegrationPoints class

`Quadrature` is the engine of the library. Implements the interface for the user and also creates the integration points for higher dimensions using a Lagrangian multiplier. It can be customized by its three template parameters which are:

- `TQuadraturePointsType` is the given integration points to create a quadrature
- `TDimension` is an unsigned integer which represents the dimension of the quadrature which by default is the dimension of the given integration points.
- `TIntegrationPointType` is the integration point type used to create a quadrature. Its default value is `IntegrationPoint` with a given dimension.

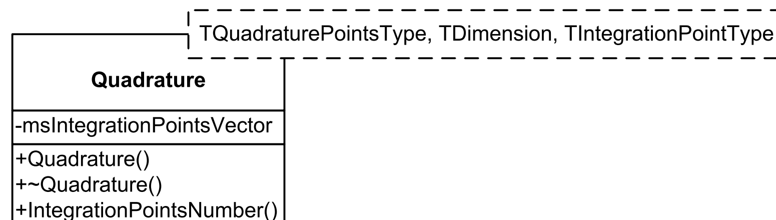

Figure 5.4: The `Quadrature` class

5.2 Linear Solvers

Implicit methods for solving finite element problems construct a system of equation representing the model and solve it with a linear solver. In this section the design of linear solver is discussed.

5.2.1 Existing Libraries

The independence nature of the solvers and also the essential and massive use of them in many analyzing methods motivates several groups to create solvers library. In Fortran there are successful libraries which are really matured and are widely used.

The *Linear Algebra PACKage (LAPACK)* is a classic library for solving linear system of equations. This library provides routines for solving a simultaneous systems of equations, eigenvalue and singular value problems, and least-squares solutions of linear system of equations. This library is written in Fortran 77 and is used widely in fortran finite element codes. *LAPACK95* is an extension of this library which provides better interface using Fortran 95 features. LAPACK is designed for running efficiently in shared-memory vector and parallel processors. To be more portable it uses the *Basic Linear Algebra Subprograms (BLAS)* for its internal operations. So by configuring the BLAS for any architecture a corresponding optimized LAPACK is obtained. The lack of iterative solvers is the only restriction of using this library for solving some big problems.

The *Linear Algebra PACKage in C++, (LAPACK++)* is an C++ interface for LAPACK. The original LAPACK++ was abandoned to start its successor *Template Numerical Toolkit (TNT)* which didn't arrive to its functionality. But the original LAPACK++ was improved by another group of developers and now has several additional features. LAPACK++ wasn't used in Kratos because in time of selecting a solver LAPACK++ was in its abandoned period.

There are also several solvers implemented in C or C++ which can be used directly in a C++ program.

The *Iterative Template Library (ITL)* provides a collection of iterative solvers and a set of preconditioners to be used with them. It also provides an abstract interface to use different basic linear algebra codes like *Matrix Template Library (MTL)* or *Blitz++*. This library can also use different type of matrices and vectors thanks to this interface. The first version of Kratos was implemented using ITL. Its clear structure and interface and its flexibility to handle different type of matrices was the reason to be selected. Unfortunately its performance was depending very much on the compiler's optimizer and basically its poor performance on Visual C++ compiler made us to put it aside.

Another important library is the *Portable, Extensible Toolkit for Scientific Computation (PETSc)*. It is constructed over BLAS, LAPACK, and *Message Passing Interface (MPI)* libraries and provides a large set of parallel linear and nonlinear equation solvers. The advantages of this library are its good performance and its support for parallel processing over distributed machines. This library is written in C and there are lack of C++ features in its structure for clean encapsulating of its tasks. This library wasn't used in implementing Kratos because parallelization at that time was not the objective of Kratos and supporting it for single process computing was too complex, (compiling and debugging this library specially under Microsoft Windows was a complex task). However there is an intention to support this library for the parallelization of Kratos in the future.

5.2.2 Kratos' Linear Solvers Library

In Kratos, a linear solvers library is implemented in order to have an small but efficient set of solvers to be used without the need to link some external solvers. This library is designed to be independent from the rest of the program, efficient in its calculation and flexible in the type of matrices and vectors it uses.

Three set of classes make the linear solvers library. Solvers encapsulate the solving algorithm. Preconditioners are used by solvers and modifying the equation system for better convergence and finally perform the inverse process to get the results. Reorderers change the equation order for less bandwidth or better cache use and also do inverse permutation for results.

Spaces For Encapsulating Operations

One of the main problems to use a linear solver or incorporate a new one is the matrix and vector types they use which can be incompatible with those used by the application. In order to solve this problem another layer of abstraction is necessary in order to encapsulate matrices and vectors operations and to be used as an interface to new matrix and vector types.

The idea used here is the same as used in LAPACK and ITL but with some modifications. As mentioned before LAPACK implements the solver algorithm and uses BLAS for its algebraic operations. This structure makes it configurable for different machines. ITL also uses a somehow similar design. It has a set of functions defined in its namespace that can be overloaded by users to implement the operation over their matrices and vectors. While ITL uses templates it can also accept different type of matrices and operate them via customized operations provided by user.

The problem with the LAPACK approach is the type of matrices and vectors that cannot be changed. For ITL this problem is solved using templates but the interface doesn't provide the complete separation of different types of matrices and vectors. For example two developers can implement two interfaces by overloading the interface methods for two different types of matrices but using the same vector. In this case operations over vectors are the same in both interfaces and causes a conflict.

In Kratos this idea is improved by defining *Space* as a new concept. `Space` defines a matrix and a vector and also their operators. Encapsulating definitions and operators in one object lets different developers to implement their interfaces without any conflict. `Space` defines a simple interface with all operations which are required by linear solvers. Table 5.1 shows this interface. In the table A and B are matrices, X,Y, and Z are vectors, and α and β are scalar constants.

Method Name	**Operation**
`Size`	Returns Size of the vector.
`Size1`	Return number of rows of matrix.
`Size2`	Return number of columns of matrix.
`Resize`	Resizes the vector
`Resize`	Resizes the matrix
`Copy`	$X \rightarrow Y$
`Copy`	$A \rightarrow B$
`Dot`	$X \cdot Y$
`TwoNorm`	$\|X\|_2$
`Mult`	$A \cdot X \rightarrow Y$
`TransposeMult`	$A^T \cdot X \rightarrow Y$
`RowDot`	$A_i \cdot X$
`ScaleAndAdd`	$\alpha X + \beta Y \rightarrow Z$
`ScaleAndAdd`	$\alpha X + \beta Y \rightarrow Y$
`GraphDegree`	Number of nonzeros in given row.
`GraphNeighbors`	Columns of nonzeros in given row.

Table 5.1: Interface of `Space`

It is important to mention that not all methods defined in `Space` has to be implemented for all user defined spaces. One can implement the sufficient set of them which are used by the solvers according to their needs.

Linear Solvers

The first requirement for linear solvers is to be interchangeable with each other. While there are no best solver for all cases and in general choosing the proper solver depends also to the problem it has to solve, it is highly desirable for users to change from one solver to other in order to find the best one for their case. This feature can be provided using the Strategy pattern. Applying this pattern to linear solvers results in the structure shown in Figure 5.5.

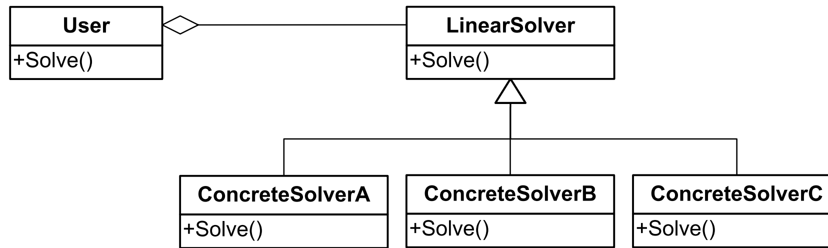

Figure 5.5: Linear solver's structure using Strategy pattern.

Using this pattern each `LinearSolver` encapsulates one solving algorithm separately and also make them interchangeable. In this way, adding a new solving algorithm is equivalent to adding a new `LinearSolver`. This encapsulation makes the library easier to extend. User keeps a pointer to the `LinearSolver` base class which may point to any member of the solver's family and use the interface of the base class to call different procedures.

This structure can be improved by dividing it into two branches, direct solvers and iterative solvers. Figure 5.6 shows this structure. Separating these two categories from each other allows special interface to be added to them.

The `IterativeSolver` class stores the tolerance, number of iterations and also the maximum number of iteration. It provides also methods for controlling the convergence and needed iterations. The `DirectSolver` is just a layer of abstraction without adding any new feature to its base class.

Linear solvers may use reorderers to reduce the bandwidth of the matrix which is important in the solution cost them using direct solvers. Reordering also can be used by iterative solvers for improving the use of cache memory. Our design must provide an easy way to add a new reordering algorithm and also let them to be combined with any linear solver.

Strategy pattern here is used to encapsulate each reordering algorithm in one class with an standard interface. In this way the extendibility is guaranteed and interchangeability is provided. Figure 5.7 shows this structure.

Bridge pattern is used to connect linear solvers and reorderers. Using this pattern users can combine each reorderers with any linear solver without problem. Figure 5.8 shows the resulted combined structure.

The next step is designing the preconditioners. They are used by iterative solvers for enhancong the convergence of the solution. Their structure must allow developers to add new preconditioning algorithms easily. It is also necessary to let users changing one preconditioner to other without problem. An strategy pattern is used in designing this structure. In this way each preconditioning algorithm is encapsulated in one object and can be added later independently to the rest of the program. Also their uniform interface established by `Preconditioner` base class allows the user to change one algorithm by another without any problem. Figure 5.9 shows this structure.

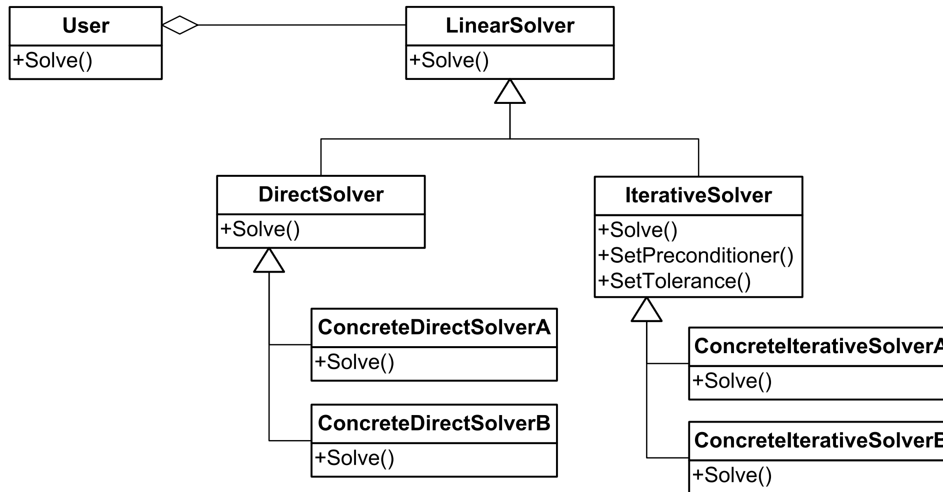

Figure 5.6: Separating direct solvers from iterative solvers.

As mentioned before preconditioners are used by iterative solvers. A bridge pattern is used to connect these two structure in a way that combining any iterative solver with any preconditioner will be possible. Figure 5.10 shows the combined structure using the bridge pattern.

All these classes take two spaces as their template parameters. One sparse space for defining operations over sparse matrices and vectors and a dense space which defines the operator over dense matrices and vectors which used as auxiliary matrices and vectors or for providing multiple right hand sides. For solving an small equation with dense matrices one can create these classes with two dense spaces without problem.

5.2.3 Examples

Here a simple conjugate gradient solver is implemented as an example to show how an algorithm can be implemented over a given space:

```
template<class TSparseSpaceType,
         class TDenseSpaceType,
         class TPreconditionerType
         class TReordererType >
class CGSolver : public IterativeSolver<TSparseSpaceType,
                                        TDenseSpaceType,
                                        TPreconditionerType,
                                        TReordererType>
  {
    public:
      CGSolver(double NewTolerance,
               unsigned int NewMaxIterationsNumber)
        : BaseType(NewTolerance, NewMaxIterationsNumber){}

      CGSolver(double NewMaxTolerance,
```

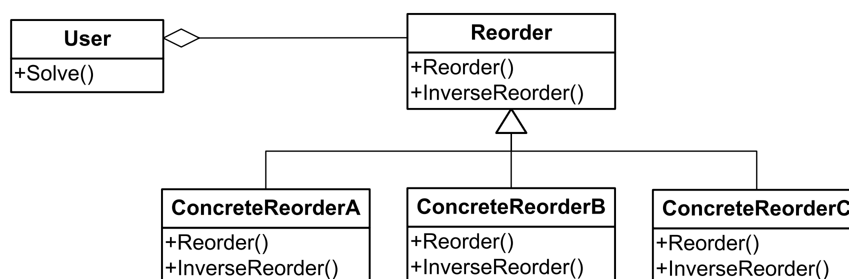

Figure 5.7: Using the Strategy pattern for designing the structure of reorderers.

```
            unsigned int NewMaxIterationsNumber,
            PreconditionerPointerType pNewPreconditioner)
    : BaseType(NewMaxTolerance,
               NewMaxIterationsNumber,
               pNewPreconditioner){}

  virtual ~CGSolver(){}

  bool Solve(SparseMatrixType& rA,
             VectorType& rX,
             VectorType& rB)
  {
    if(IsNotConsistent(rA, rX, rB))
      return false;

    GetPreconditioner()->Initialize(rA,rX,rB);
    GetPreconditioner()->ApplyInverseRight(rX);
    GetPreconditioner()->ApplyLeft(rB);

    bool is_solved = IterativeSolve(rA,rX,rB);

    GetPreconditioner()->Finalize(rX);

    return is_solved;
  }

private:

  bool IterativeSolve(SparseMatrixType& rA,
                      VectorType& rX,
                      VectorType& rB)
  {
    const int size = TSparseSpaceType::Size(rX);

    mIterationsNumber = 0;

    VectorType r(size);
```

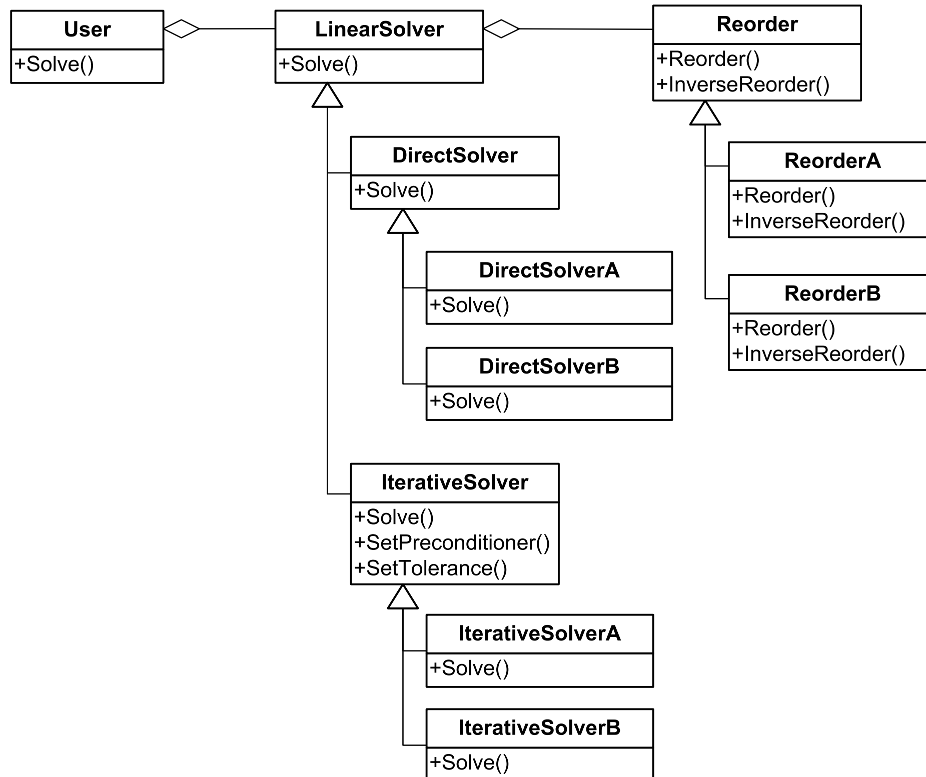

Figure 5.8: Applying the bridge pattern for connecting linear solvers and reorderers.

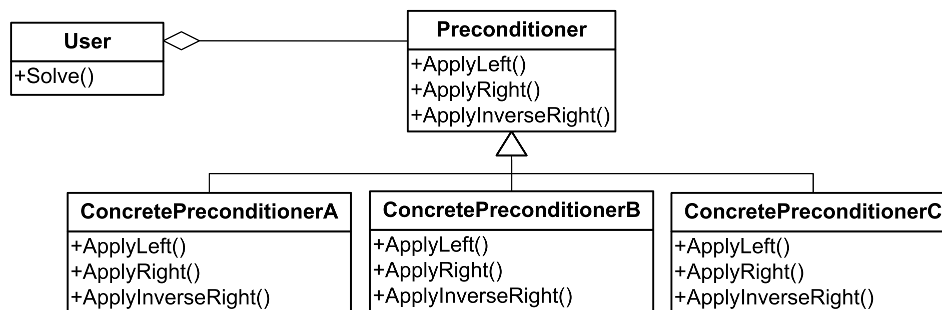

Figure 5.9: Strategy pattern is used for designing the structure of preconditioners.

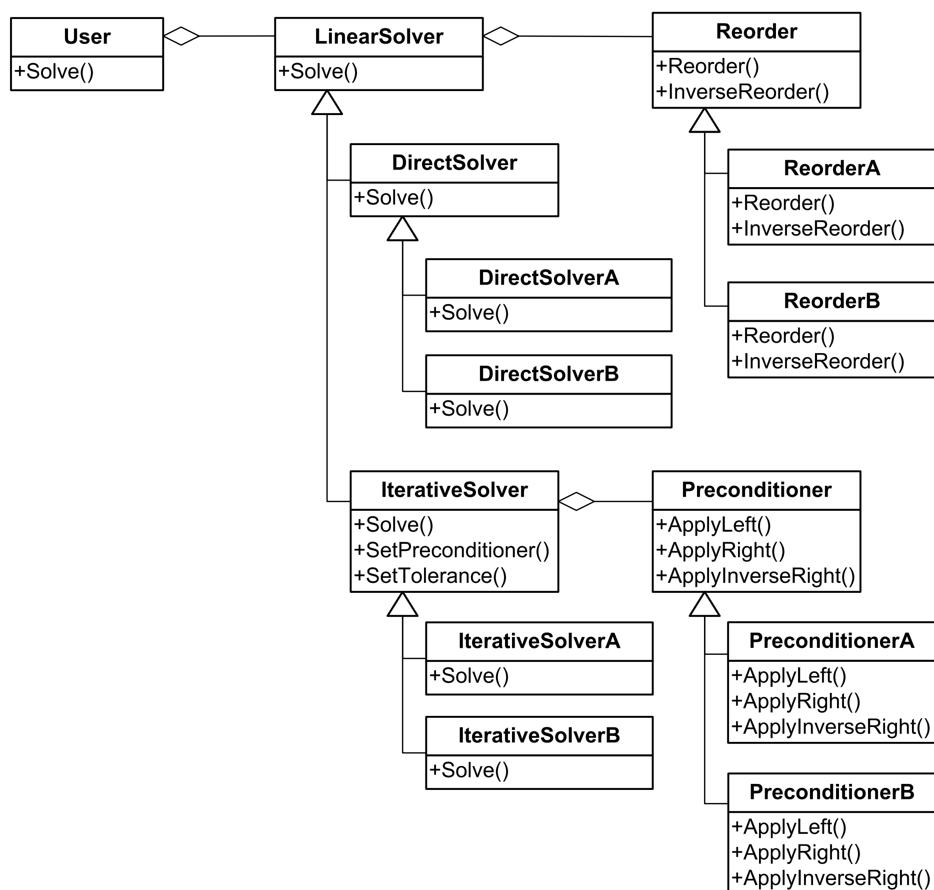

Figure 5.10: Applying the bridge pattern in connecting iterative solvers and preconditioners.

```
PreconditionedMult(rA,rX,r);
TSparseSpaceType::ScaleAndAdd(1.00, rB, -1.00, r);

mBNorm = TSparseSpaceType::TwoNorm(rB);

VectorType p(r);
VectorType q(size);

double roh0 = TSparseSpaceType::Dot(r, r);
double roh1 = roh0;
double beta = 0;

if(roh0 == 0.00)
return false;

do
{
  PreconditionedMult(rA,p,q);
```

```
        double pq = TSparseSpaceType::Dot(p,q);

        if(pq == 0.00)
          break;

        double alpha = roh0 / pq;

        TSparseSpaceType::ScaleAndAdd(alpha, p, 1.00, rX);
        TSparseSpaceType::ScaleAndAdd(-alpha, q, 1.00, r);

        roh1 = TSparseSpaceType::Dot(r,r);

        beta = (roh1 / roh0);
        TSparseSpaceType::ScaleAndAdd(1.00, r, beta, p);

        roh0 = roh1;

        mResidualNorm = sqrt(roh1);
        mIterationsNumber++;
      } while(IterationNeeded() && (roh0 != 0.00));

      return IsConverged();
    }

}; // Class CGSolver
```

5.3 Geometry

Solving a problem using finite element method requires the discretization of model which consist of dividing the model into several small partitions with a defined shape. Geometries are designed to encapsulate these shapes, manage their data, and calculate their parameters. In this section the design and implementation of geometries in Kratos is described.

5.3.1 Defining the Geometry

As described in section 3.2 in the FEM a partial differential equation is converted to its equivalent integral form in discrete space which has a general form:

$$\int_{\Omega} L_1(Na)d\Omega + \int_{\Gamma} L_2(Na)d\Gamma = 0$$

where N is the shape function and a is the interpolated parameter. This integral form is divided into sub integrals over domain and boundary and which results in the following elemental form:

$$\sum_{e}^{n_e} \int_{\Omega_e} L_1(Na)d\Omega_e + \sum_{c}^{n_c} \int_{\Gamma_c} L_2(Na)d\Gamma_c = 0$$

where n_e is the number of elements and n_c number of conditions. This form consists of integrals over elements or boundary domain. The value of the shape functions N and their derivatives are necessary for evaluating the integrals as operators L_1 and L_2 usually have derivatives operators.

The derivatives of shape functions are computed via the inverse of the jacobian matrix **J** as it can be seen in following equations for the three dimensional space:

$$\begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \\ \frac{\partial N_i}{\partial \zeta} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{bmatrix} \begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix} = \mathbf{J} \begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix}$$

$$\begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix} = \mathbf{J}^{-1} \begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \\ \frac{\partial N_i}{\partial \zeta} \end{bmatrix}$$

To perform the integration one way is to change from global coordinates to local ones. This transformation requires the calculation of the jacobian matrix J and its determinant. All this process is described to define the geometry and help us to identify what can be encapsulated by the geometry.

`Geometry` encapsulates all the geometrical information like its dimension, points, edges and also some auxiliary operations like calculating the center point, area, volume, etc. Beside this geometrical information, it also provides the integration points and also the value of the shape functions and their local and global derivatives. It also calculates the jacobian matrix **J** , the inverse of the jacobian $\mathbf{J}^{-1}$ and its determinant $|\mathbf{J}|$.

All these operations are encapsulated to enable the element developers to write their `Element`s in a generic form and independent from the type of geometry.

5.3.2 Geometry Requirements

First of all `Geometry` has to be very lightweight and memory efficient. This requirement comes form the fact that modeling a real problem usually needs to store a large number of geometries in memory and process them. Having any unnecessary or even less necessary overhead for each geometry, results a significant overhead in memory used by program.

`Geometry` also has to be fast in its operations. `Element`s use geometries to perform calculation in most inner loops of the code, so their performance severely affects the global performance of the code specially in time of building equations system. In this way all parameters that can be calculated once must be kept to be used without recalculations. Also interfaces and algorithms must be designed and implemented in a way that minimum creation of temporaries be necessary.

Another requirement is enabling users to combine a generic `Element` with any `Geometry` without changing it. To achieve this objective the geometry structure has to provide an standard interface for all geometries. Doing this changing from one geometry to other will be easy while the interface to be used will be the same.

Finally adding a new geometry must be done without any need to change other part of the code. A good encapsulation and clear connection to other part of the code can help to achieve this objective.

5.3.3 Designing Geometry

Like in the linear solver's structure, an strategy pattern is used in the geometries structure. Using this pattern makes them interchangeable via a uniform interface. It also encapsulates all `Geometry` data and operations in one object which makes them more independent from each other and easier to be extended. Figure 5.11 shows this structure.

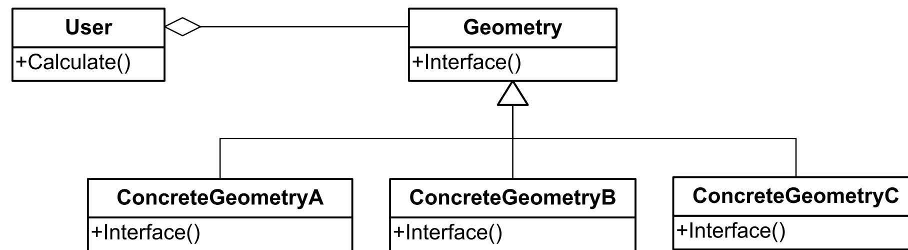

Figure 5.11: Designing the geometries structure, using the strategy pattern.

A composite pattern also can be used to let users combine different geometries to form a complex geometry. This structure may be useful in some situations where a complex geometry has to be defined. Figure 5.12 shows proposed structure.

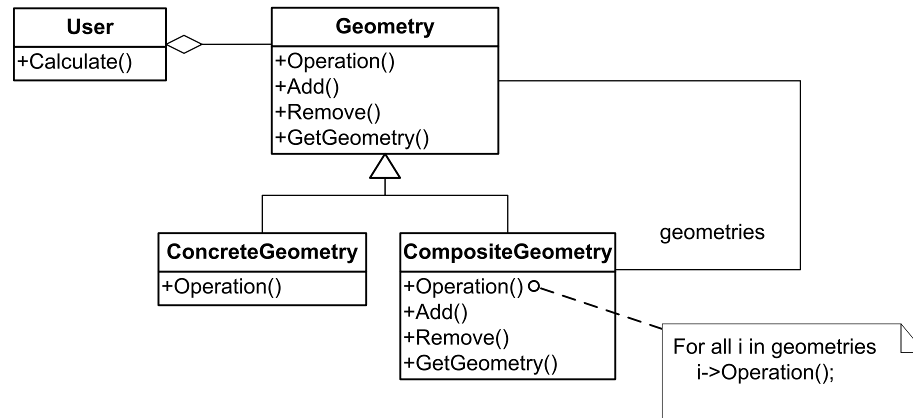

Figure 5.12: Composite pattern lets users combine different geometries in one complex geometry.

There are different ways of implementing geometries. Some common ways are to defined them as a set of points and constructive rules. The other way is an hierarchy form in which a three dimensional shape is defined by its two dimensional edges. These edges like any other two dimensional geometry are defined by their 1 dimensional edges. Finally these edges are defined by points. Geometries implemented in this form provide a complete set of information to users and are useful in situations that detailed information about the geometry and its edges is required.

In Kratos this implementation is considered to be too sophisticated and the first approach is used. There are two main reason for this decision. Algorithms implemented in Kratos work with geometry points (`Nodes` for `Element` geometry). For this reason a fast access to `Nodes` is more important than to its edges. The second reason is the memory overhead for holding all hierarchy is considerable and redundant for usual elemental algorithms. All these made us to use the first approach but keeping an empty place in geometry for inserting edges data base. `Geometry` is derived from points array. This approach lets users to operate with geometry like an array of points. For example move it just by moving all points in a loop or applying C++ standard library

Method	Information
`Dimension`	Dimension of the geometry
`WorkingSpaceDimension`	Dimension of space which geometry is definied
`LocalSpaceDimension`	Geometries local dimension
`PointsNumber`	Number of points creating geometry
`Length`	Length of 1 dimensional geometries
`Area`	Area of 2 dimensional geometries
`Volume`	Volume of 3 dimensional geometries
`DomainSize`	Length, area, or volume depending to dimension
`Center`	Center point of geometry
`Points`	Geometries' points
`pGetPoint`	Pointer to i-th point of geometry
`GetPoint`	i-th point of geometry
`EdgesNumber`	number of edges of this geometry
`Edges`	Edges of geometry
`pGetEdge`	Pointer to i-th edge of geometry
`GetEdge`	i-th edge of geometry
`IsSymmetric`	True if geometry is symmetric

Table 5.2: `Geometry` methods implementing geometrical operations.

to it using its point iterator.

The interface of `Geometry` reflects all operations provided by it. Table 5.2 shows methods implementing geometrical operations and table 5.3 shows methods providing finite element operations.

In order to optimize the performance of `Geometry` all interface which produces results in form of vector, matrix, or tensor accept their results as an additional argument. Here is an example:

```
Matrix& Jacobian(Matrix& rResult,
                 IndexType IntegrationPointIndex) const
{
  // Calculating the jacobian...

  return rResult;
}
```

The alternative design is getting parameters as arguments and return the calculated results. Here is the previous example using this alternative design:

```
Matrix Jacobian(IndexType IntegrationPointIndex) const
{
  Matrix result;

  // Calculating the jacobian...

  return result;
}
```

This alternative seems to be more elegant but produces a significant overhead in `Geometry` performance due to creating several unnecessary temporaries.

The information provided by `Geometry` can be divided in two different categories:

Method	Information
`HasIntegrationMethod`	True if implements the integration method
`IntegrationPointsNumber`	Number of integration points
`IntegrationPoints`	Array of integration points
`GlobalCoordinates`	Global coordinates related to given local one
`Jacobian`	Jacobian matrix $\mathbf{J}$
`DeterminantOfJacobian`	Determinant of jacobian $\lvert\mathbf{J}\rvert$
`InverseOfJacobian`	Inverse of jacobian $\mathbf{J}^{-1}$
`ShapeFunctionsValues`	shape functions' values in integration points
`ShapeFunctionValue`	Value of shape function i in integration point j
`ShapeFunctionsLocalGradients`	
`ShapeFunctionLocalGradient`	
`ShapeFunctionsFirstDerivatives`	
`ShapeFunctionsSecondDerivatives`	
`ShapeFunctionsThirdDerivatives`	

Table 5.3: `Geometry` methods providing finite element operations.

Constants Information which depends only to the type of geometry and are equal for all geometries with the same type. Dimension, integration points, shape functions values, and shape functions local gradients are examples of operations in this category.

Nonconstants The second category contains the rest of information which can change from one geometry to another. For example points, jacobian, and shape functions gradients are in this category.

In order to optimize even more the performance of geometry, the different nature of these two categories must be considered. First, constant information can be calculated once and stored to be used if there are necessary. Also there is no need to access this information using virtual methods. Virtual functions can reduce the performance due to their function call delay specially for methods with small operations like returning a value. So making these methods no virtual can increase their performance.

As mentioned earlier constant information can be calculated once and stored to be used later. The draw back of this idea is the memory required by pointers pointing to these data for each geometry. While the memory used by these pointers is relatively small, but creating a large number of geometries makes it considerable. A good solution to this problem is creating an auxiliary object and put all constant information in it. In this way only one pointer in each geometry is necessary to point to this object and access all constant information. `GeometryData` is defined and implemented to hold all constant information in geometry. Figure 5.13 shows this class and its attributes.

`Geometry` keeps a reference to the geometry data and use it to provide constant information by its no virtual methods. Figure 5.14 shows the complete structure.

For each type of geometry an static variable of `GeometryData` is created and its reference is given to geometry in construction time.

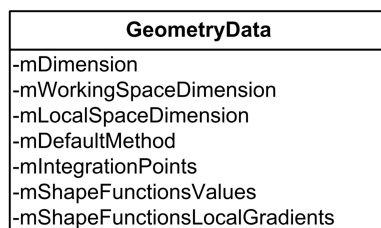

Figure 5.13: `GeometryData` class and its attributes.

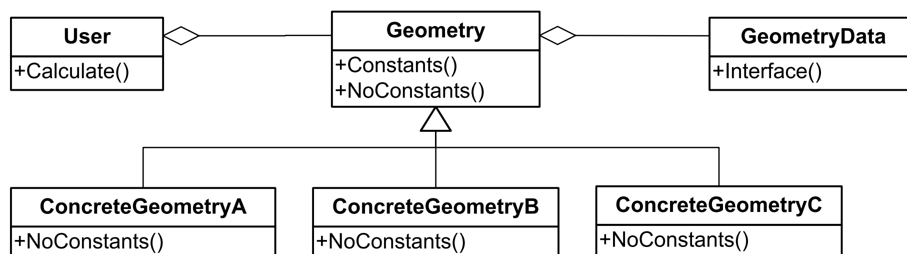

Figure 5.14: `Geometry` uses `GeometryData` for providing constant information and its derived classes only implement the rest of operations.