SPE 36007

FLEX: An Object-Oriented Reservoir Simulator

Santosh Verma, SPE, and Khalid Aziz, SPE, Stanford University

This paper was prepared for presentation at the Petroleum Computer Conference held in Dallas, Texas, 2-5 June 1996.

Abstract

This paper describes the design of FLEX, an object-oriented, flexible grid, black-oil reservoir simulator. An object-oriented decomposition of a reservoir simulator helps in dealing with the complexity of this problem. This approach is particularly useful because of the difficulties associated with generation and use of flexible grid geometries (like Voronoi, median, boundary adapting grids, etc.).

The entire problem is divided into subsystems like geometry, gridnodes, gridnode connectivity, grid, reservoir fluid flow, and matrix. Each of these subsystems have objects which are closely related. The dependency of these subsystems is established. A detailed analysis of each subsystem leads to identifying the classes, which are a set of objects having similar behavior. Attributes and behavior of the classes are assigned. After establishing relationships between the classes, they are arranged into hierarchies. About one hundred major classes have been identified and designed to achieve the desired behavior from FLEX. The programming language used is C++.

Introduction

Reservoir simulators are inherently complex. A simulator has to deal with issues such as reservoir and grid geometry, fluids, flow calculation, matrix computations, several well and production constraints, visualization, etc. The most important feature of FLEX, a black oil simulator, is its ability to handle complexities arising from flexible grids. Verma and Aziz (1996) give a description of flexible grids in reservoir simulation. The flexibility in grids increases geometrical complexities as well as complexities in flow calculation. These complexities need sophisticated data structures (and associated procedures) to simplify the problem. It is expected that FLEX will change with time to incorporate new features. One of the important considerations in designing the simulator is the ease with which the simulator can be expected to handle new problems. All these factors combined to make the development process of FLEX quite complex. This paper describes the advantages of using an object-oriented approach for the development of reservoir simulators. The philosophy followed in designing FLEX is that advocated by Booch (1994) and Cheriton (1995).

Basic Features of FLEX

FLEX solves flow equations based on the control volume formulation (see Verma and Aziz, 1996). It uses the Newton-Raphson method to iteratively solve for the variables. A connection-based approach is employed to form the Jacobian matrix and the residual vector (see Lim, Schiozer and Aziz, 1995 and Verma and Aziz, 1996). Presently the simulator is developed to handle only two immiscible phases.

The gridnodes can be located so that they represent reservoir geometry, wells, faults, etc. Figure 1 is an example of the flexible grid generation capabilities of FLEX.

Why Object-Oriented?

An object-oriented approach was followed in the design of FLEX to handle the complexities associated with a flexible grid simulator, and to provide for future enhancements.

What is an object? An object is defined as a programming entity that encapsulates both state and behavior behind an interface (Cheriton, 1995). In C++, the state of an object is represented by its data members and the behavior by its member functions. Because of such a coupling, objects ensure data integrity. The state and behavior are coupled using the mechanism of *class*. A class describes the state and behavior which are common to objects of that class. For example, in C++, a simple matrix class can be defined as:

```
class Matrix {
public:
  Matrix() ;
  Matrix(int nr, int nc, double *mat) ;
  ~Matrix() ;
  Matrix operator+(const Matrix & B) ;
  //Many more functions defined.
protected:
```

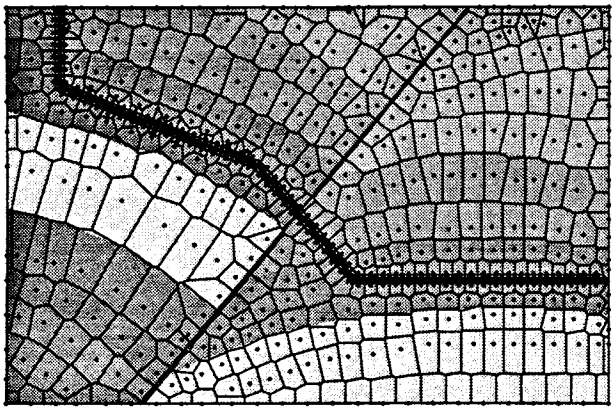

Figure 1: Voronoi grid along deviated well, layer boundaries and fault

```
  int NumRows_;
  int NumColumns_;
  double *Matrix_;
};
```

Its attributes (i.e. data members) are number of rows and columns, and the entries of the matrix. The + operator is overloaded (the + operator is said to be overloaded when it is defined to add two objects) to add two matrices. Thus if A and B are two matrices then they can be added, using the expression C=A+B, to obtain matrix C.

Complex Systems and Reservoir Simulators. Reservoir simulators are complex systems that have some basic attributes. Normally a complex system has a hierarchy wherein it is composed of interrelated systems that have in turn their own subsystems and so on until some elementary component is reached. Normally the intracomponent linkages are stronger than the intercomponent linkages (Booch, 1994). These elementary components can be arranged in a hierarchy. Hierarchy in a complex system normally takes two common forms. An "is a" hierarchy and a "has a" hierarchy, e.g., a sparse matrix "is a" matrix, a Voronoi gridblock "is a" gridblock, while grid "has a" number of gridblocks. Such relationships were identified in FLEX during the object-oriented analysis stage.

Dealing with Complexity: Decomposition. As human beings we deal with a complex system by decomposing it into smaller and smaller parts. The decomposition can be done in two ways: *algorithmic* and *object-oriented.* In algorithmic decomposition (AD) each module in the system (e.g. a simulator) denotes a major step in some overall process. While in object-oriented decomposition (OOD) the system (e.g. a simulator) is seen as a set of independent *objects* that interact to perform some higher level behavior. Since we decompose our problem using objects we call this *object-oriented decomposition.* Of course, it is not possible to represent all the behavior of a system only by objects. Algorithmic aspects have to be included also. The resulting structure obtained from OOD can be used to express the algorithmic viewpoint.

Advantages of Object-Oriented Decomposition (OOD). Languages such as Fortran deal with complex sytems by explicit support of algorithmic decomposition of a problem. Thus the program is divided into several routines each of which does a specific job. There are limits to the amount of complexity that can be easily handled using algorithmic decomposition. On the other hand some languages explicitly support object-oriented decomposition of the problem, by which the complex system is decomposed into interacting objects. What are the advantages of OOD? Booch (1994) points out that OOD has several significant advantages over algorithmic decomposition when dealing with complex systems. As he observes, "the complex system can be decomposed into much smaller systems through the reuse of common mechanisms, thus providing an important economy of expressions". Figure 2 shows a piece of code written to solve a linear system of equations using the conjugate-gradient residual algorithm. The language used is C++. The left column in the figure shows the algorithm while the right column shows the code. It illustrates the use of Matrix and Vector objects in a procedure. The economy of expressions possible using OOP is quite evident from the figure. Since object-oriented systems are based on smaller intermediate systems (in which we already have confidence), they are more resilient to change. Since the interactions between the Matrix and Vector objects are well defined and tested, the procedure can be modified (e.g. to include preconditioning of matrix **A**) with confidence. Furthermore these objects can easily evolve over time, greatly reducing the risk of building complex systems. Thus, for example, the same procedure can also be used with matrix objects which belong to a class derived from the Matrix class, since object-oriented programming (OOP) provides mechanisms to derive a class, (called *derived*) from another class (called *base*), wherein, a derived class inherits the attributes and behavior of the base class and adds to it its own attributes and behavior.

Steps in Object-Oriented and Design (OOD)

The identification of classes and objects is the fundamental issue in OOD (the C++ terminology used in this section can be found in any text on C++, e.g. Lippman, 1991.) OOD requires the following to be done:

- Identify classes
- Assign attributes and behavior to classes
- Find relationships between classes
- Arrange classes into hierarchies

A good guidline for designing object types is to look for nouns and verbs in the description of the problem to be solved. Typically nouns would become classes in the program design and the verbs become the operations (member functions). Once the classes have been identified, the next step is to find what information should an object of that class contain (data members) and the operations that an object can perform or the operations that can be performed on that object (member functions). Then, one has to find the relationships between the classes. Some of

	Vector x1, r0, r1, p0, p1; Vector Ar0, Ar1, Ap0;
	double alpha, beta; double EPS = 1.E-10;
Choose $\vec{x}_0$	**Vector x0; int count = 0; int MAX=30;**
Compute $\vec{r}_0 = \vec{b} - A\vec{x}_0$	**r0 = b - A*x0;**
Set $\vec{p}_0 = \vec{r}_0$	**p0 = r0;**
	Ar0 = A*r0;
For i = 0, 1, 2, ..., compute	**while((r0.Norm()>EPS) && (count<MAX))**
	{
$\alpha_i = \frac{\vec{r}_i^T(A\vec{r}_i)}{(A\vec{p}_i)^T(A\vec{p}_i)}$	**Ap0 = A*p0;**
$\vec{x}_{i+1} = \vec{x}_i + \alpha_i\vec{p}_i$	**alpha = r0*Ar0/(Ap0*Ap0);**
$\vec{r}_{i+1} = \vec{r}_i - \alpha_i A\vec{p}_i$	**x1 = x0 + alpha*p0;**
	r1 = r0 - alpha*Ap0;
$\beta_i = \frac{\vec{r}_{i+1}^T(A\vec{r}_{i+1})}{\vec{r}_i^T(A\vec{r}_i)}$	**Ar1 = A*r1;**
$\vec{p}_{i+1} = \vec{r}_{i+1} + \beta_i\vec{p}_i$	**beta = r1*Ar1/(r0*Ar0);**
	p1 = r1 + beta*p0;
	p0 = p1; r0 = r1; Ar0 = Ar1; count++
	}

Figure 2: OOP Example: Conjugate Residual Algorithm

the classes may remain independent of other classes, but others will depend upon the existence of other classes. There may be some classes which contain other classes (composition of classes) as data members. Some classes may use other classes as their member classes while some may be special cases of another class (inheritance of classes). Once the class names, their attributes and behavior have been decided and class relationships established, it becomes easier to recognize which classes can inherit data members and functionality from other classes. Typically composition should be used when one class "has" another class, while inheritance should be used when one class "is" a kind of another class. In the description of class diagrams presented later in this paper a number of examples of such classes are given.

OOD is often an iterative process, requiring several attempts at exact problem description, identification of abstractions and testing implementations. There are several issues which have to be considered when designing the classes. A few, among those which were considered in designing FLEX, are:

- How are the data members, member functions, etc. named? There should be consistency in naming conventions of data members and member functions.
- Should the data members of a class be public, protected or private? If a data member is public, it can be directly accessed and manipulated by any function. If it is private it cannot be directly accessed by any object or function. Also, it cannot be used by a derived class, i.e., it can not become a base class. One should normally make data members protected so that the class can serve as a base class.
- Should copy constructors and destructors for the classes be defined? A copy constructor is used to copy one object to another of the same or derived type. A destructor is used to destroy an object when it is no more required. C++ provides default constructors and destructors. Typically, before defining, one should ask if this is really needed.
- How many levels of inheritance is good? The objective should be to allow the user to follow through the hierarchy with relative ease. Typically, three to four levels of inheritance is optimum.
- Should multiple inheritance be allowed? Multiple inheritance means deriving a class from more than one base class. Multiple inheritance should be avoided unless necessary.
- Should a function be defined virtual, nonvirtual or pure virtual in the base class? When a function is declared virtual in the base class then a function in the base class can be redefined in the derived class. The version to use is determined at run time and depends on the type of object invoking that function. This process of choosing which version of a function to use while the program is running is called dynamic linking. If a virtual function does not have a default implementation then it is called a pure virtual function and a derived class has to define that function. A nonvirtual function should not be redefined in the derived class. It is good to make functions virtual, though their efficiency may become a criterion.
- Should a class be declared a friend of another class? Issues of friendship (i.e. which class has direct access to private/protected data members of other class) should be handled with care. Its good to avoid friendship unless really necessary.
- What is the best way to implement an abstract base class (ABC)? A base class which has no implementation is called an abstract base class. It is used to specify inter-

face for derived classes. One way to implement an ABC is by making one of its member functions pure virtual. Another way is to make its constructors protected. Its better to make constructors protected rather than define a pure virtual function to make an abstract base class. It is also good to have the maximum member functions possible in the base class so that all possible functions of the base class are defined. The data members need not be complete and they can be added in the derived classes.

- Should templates be used? Templates provide a mechanism for indicating types that need to change with each class instance. Implementing templates in compilers is relatively new and they typically make code less portable. Hence, they should be avoided.

Design of FLEX

Requirements from FLEX. FLEX should be able to provide the user with various reservoir state parameters, as a function of time, at specific points (called gridnodes) of the reservoir domain in three-dimensions. Specifically, it must give phase saturations and pressure. One should be able to arbitrarily locate nodes to represent reservoir geometry, local flow geometry, faults, etc. The user should be able to construct gridblocks (in two- and three-dimensions) around these nodes based on a number of different criteria, e.g. Voronoi, generalized perpendicular bisector (GPEBI), control volume finite element (CVFE or median), mixed Voronoi-CVFE and Cartesian. Verma and Aziz (1996) give a description of these grids. The computation of flow between nodes, i.e. along connections, should take into account grid block geometry, full, asymmetric and anisotropic permeability tensor and associated control volume rock and fluid properties. It should be able to handle boundary conditions like wells, aquifers and no-flow boundaries. Relevant information to view the grids must also be available to the user. The solution of state parameters require solution of a large system of equations using linear solvers. Matrix computations should be easy to handle. The software architecture of FLEX should be able to evolve over time and adapt to changing requirements.

Object-Oriented Design of FLEX. Knowing what the requirements are from FLEX, we attempt its object-oriented design. As mentioned earlier, the identification of classes and their design is the fundamental issue in OOD. Before identifying the classes, the system is broken into a number of subproblems. A study of the requirement for the FLEX reservoir simulation system suggests that the problem can be broken down into the following four groups:

- Location of nodes in the reservoir domain.
- Generation of grid blocks around nodes.
- Computing flow (with specifiied different boundary conditions) along the network of connections. The flow derivatives with respect to the unknowns are also to be computed.
- Solution of the linear system of equations using matrix operations to determine changes in state parameters with time.

Based on these subproblems, FLEX is broken into seven subsystems. They are **Utility, Geometry, Matrix, GridNode, Connectivity, Grid,** and **Reservoir** subsystems.

The dependencies of these subsystems are given in Figure 3. The **Utility** subsystem consists mainly of independent classes which deal with some of the very basic objects, e.g., arrays. The **Geometry** subsystem contains a number of related classes which deal with the fundamental geometrical issues in the system. The classes in the **Matrix** subsystem mainly deal with matrix computations and other special matrices which may arise in FLEX. It is dependent on the **Utility** subsystem only. The classes in the **Utility, Geometry** and **Matrix** subsystems have global use, i.e., any class in the remaining subsystems may use the classes in these three subsystems.

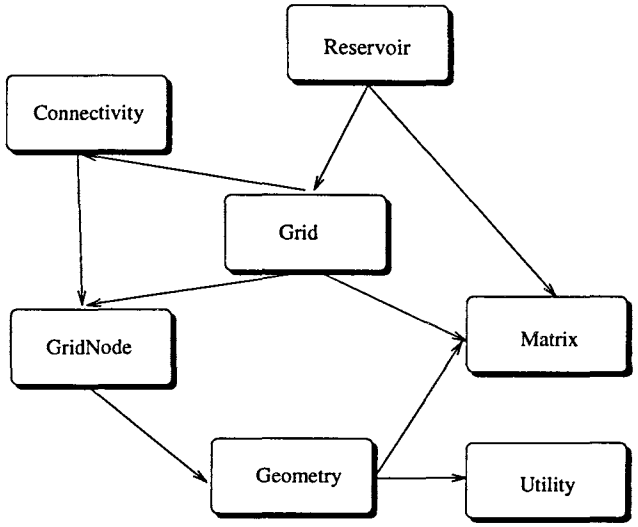

Figure 3: FLEX system top-level module diagram

The **GridNode** subsystem contains classes which colloborate to give a set of **GridNodes**. The classes in this subsystem are dependent on the **Geometry** and **Utility** subsystems. The gridnodes are triangulated to give node connectivity. The classes in the **Connectivity** subsystem describe the connectivity information arising out of the process of triangulation. They determine various connectivity relationships among the fundamental geometrical elements, i.e., tetrahedron, triangles, edges and nodes, occurring in the system. They are dependent on the **Grid Node** subsystem.

The **Grid** subsystem has a number of classes which represent various types of grids used FLEX. This subsystem is dependent on the **Connectivity** and **GridNode** subsystems. The **Reservoir** subsystem is directly dependent on the **Grid** subsystem and the **Matrix** subsystem. It is the most active subsystem in FLEX. Various classes in this subsystem colloborate to react to different boundary conditions in the reservoir and give state parameters as functions of space and time.

In the next few sections the subsytems and their classes are discussed through class diagrams. A single class diagram represents a view of the class structure of a system. The two essential elements of a class diagram are classes and their basic relationships. Some of the possible connections among classes are association, inheritance, "has," and "using" relationships. All the notations used in the class diagrams are taken from Booch (1994) and are shown in Figure 4.

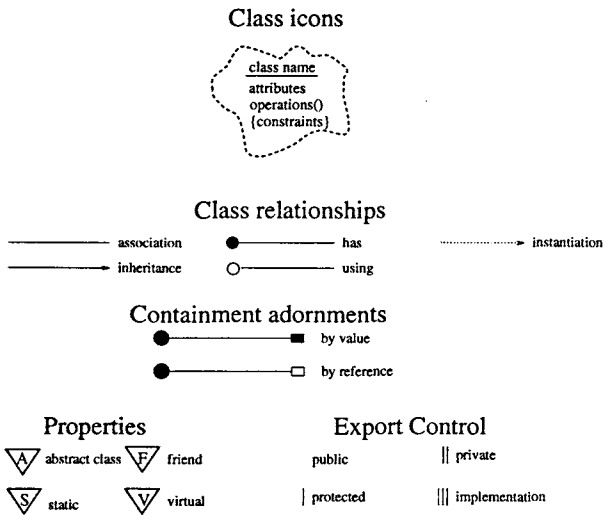

Figure 4: Notations for class diagram: after Booch (1994)

Utility classes. This subsystem contains some of the basic classes. They are

- Array: An array
- DArray: An array of doubles
- IArray: An array of integers
- String: An array of characters
- StringArray: An array of strings
- Word: An array of characters except blanks
- LinChar: An array of Words

Arrays are good candidates for a template. Templates may have portability problems, hence they are not used in FLEX. Instead, a class called Array is defined which has functions common to all the array classes. It is defined as an abstract base class by making its default constructor protected. The definition of this class is given below:

```
class Array {
public:
  int Size();
  int Size() const;
  virtual ~Array();
  virtual Array &operator=(const Array &A);
protected:
  int Size_;
  Array(const Array &A);
  Array() ;
  Array(int size) ;
  void CheckIndex(int i);
  void CheckIndex(int i) const;
};
```

Data member Size_ represents size of the array. The assignment operator = and destructor ~Array() are made virtual because they should be redefined in the derived class. CheckIndex is a nonvirtual function because it has a common implementation among all the array types and should not be redefined in the derived classes.

All the other array classes are derived from the Array class. The relationship between these array classes is shown in Figure 5. DArray, IArray, String and Word use built-in data types. A StringArray "has a" a number of strings while a LineChar "has a" number of Words. Thus StringArray and LineChar classes are "composite" classes because they contain instances of objects of other classes. A number of procedures which are specific to these classes are defined as member function implementation of these classes, e.g., the overloaded + operator to add two Strings. The definition of the DArray class, which is derived from Array class, is shown below:

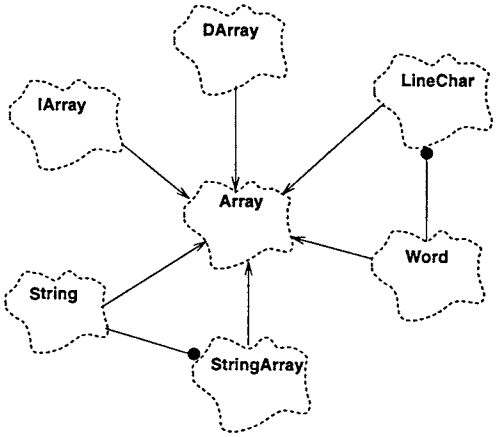

Figure 5: Class diagram of **Utility** classes

```
class DArray:public Array {
public:
  DArray();
  DArray(int n, double *val);
  DArray(const DArray & A);
  ~DArray();
  double & operator[](int i);
  const double & operator[](int i)const;
  DArray & operator=(const DArray & A);
  friend DArray operator*(double a, const DArray & A);
  friend DArray operator*(const DArray & A, double a);
  friend DArray operator+(const DArray & A, const DArray & B);
  friend DArray operator-(const DArray & A, const DArray & B);
  friend DArray operator/(const DArray & A, double a);
  friend DArray operator-(const DArray & A);
protected:
  double *Arr;
};
```

It redefines the virtual functions of the Array class. A number of other functions are also defined. A number of operators (like *, +, -) are overloaded to do arithmetic calculations with DArray objects.

Matrix classes. The matrix classes are used in solving the linear system of equations which arise in FLEX. The classes in this subsystem are shown in Figure 6. The Matrix class is an abstract base class. It has the member functions common to all the derived matrix classes. IFullMatrix class is defined for matrices which are full and have integer entries. In a similar manner, DFullMatrix class is defined for full matrices with entries of type double. Both the classes are derived from Matrix. The additional data member of IFullMatrix class is an integer array and for DFullMatrix is a double array. A number of matrix operations are defined for both these classes. An ISparseMatrix is derived from IFullMatrix matrix. It is considered to be a special case of IFullMatrix matrix. A number of entries of this matrix are zero, hence they need not be stored. To model the special behavior a few data members are required. These data members store information on the matrix sparsity. A similar derived class for DFullMatrix class called DSparseMatrix is also defined with similar additional data members. A Jacobian class is derived from DSparseMatrix class. A few classes such as XRotationMatrix, YRotationMatrix and ZRotationMatrix are derived from DFullMatrix class. These matrices differ from DFullMatrix in the way they are constructed and are used for computing rotations of points and lines about an axis. Since a permeability tensor can be full, a class called PermTensor is derived from DFullMatrix class. The Vector class is derived from the DArray class. A number of operators for the Matrix and Vector classes are defined which allow compact representation of procedures as shown in Figure 2.

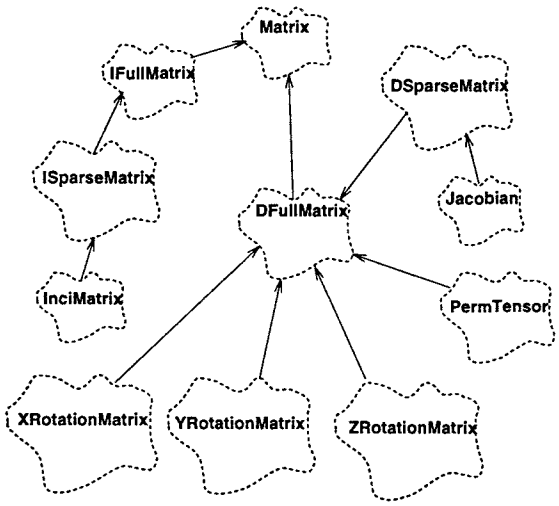

Figure 6: Class diagram of **Matrix** classes

The DFullMatrix class is defined below:

```
class DFullMatrix:public Matrix {
protected:
  double *Mat;
public:
  DFullMatrix();
  DFullMatrix(int rows, int cols, double *mat);
  virtual DFullMatrix(const DFullMatrix & FM);
  ~DFullMatrix();
  virtual DFullMatrix operator~()const;
  virtual DFullMatrix & operator=(const DFullMatrix &A);
  virtual DFullMatrix operator->*(const DFullMatrix & B);
  virtual DFullMatrix operator+(const DFullMatrix & B);
  virtual DFullMatrix operator-(const DFullMatrix & B);
  virtual DFullMatrix operator-()const ;
  virtual DFullMatrix operator*(const DFullMatrix & B);
  virtual Vector operator*(Vector & V);
  virtual DFullMatrix operator/(const DFullMatrix & B);
  virtual double & operator()(int i, int j) ;
  virtual const double & operator()(int i, int j)const;
  friend DFullMatrix operator*(double a, const DFullMatrix & B);
  virtual double AbsColumnSum(int c);
  virtual double ColumnMaxNorm(int c);
  virtual Vector DirectSolve (VVector &b);
  virtual DFullMatrix Inverse();
};
```

This class has overloaded operators to compute the transpose of a matrix (operator ~), matrix product (operator $->*$), matrix addition (operator +), matrix subtraction (operator -), etc. A member function, DirectSolve, to solve a system of equations is also defined. The DSparseVmatrix class which inherits functions and data members from DFullMatrix class redefines those which are specific to sparse matrices.

Geometry Classes. Constructing the gridblock geometry requires complex geometrical calculations. The **Geometry** classes defined in this subsystem simplify these calculations with the use of objects. The **Geometry** subsystem contains many classes which interact to produce the desired behavior for constructing gridblock geometry. Some of them are:

- Point: Its attributes are the x, y and z location of a point.
- PointSet: A set of points.
- SpaceVector: It represents a vector in space and is derived from the Vector class. It has a maximum dimension of three. Number of geometry related functions are defined in this class.
- Line: A line has a Point and a SpaceVector as attribute. The Line object passes through the Point object in the direction given by a SpaceVector object.
- LineSegment: A line segment given by the two Points it connects.
- Polyline: A line passing through many points.
- StreamLine: A stream line.
- StreamLineSet: A set of streamlines.
- Plane: A plane has a Point and a SpaceVector as attributes. It passes through the Point object. The SpaceVector object represents the normal to the plane.
- Edge: It represents connection between two gridnodes.
- Face: A face of a gridblock. Its attributes are a set of points. These points are at the vertices of the face.

These classes are used for most of the geometrical calculations. Their relationships are shown in Figure 7. Most of the relationships shown in the figure are of "use" and "has a" type. Thus, e.g., a Line has a SpaceVector and a Point object as its data member. Similarly, e.g., a Plane has a SpaceVector and a

Point object as its data member. A Face object has a number of Points and a PointSet object has a number of Points. A Polyline class is derived from a PointSet since it is a set of Points and represents a line that passes through these points.

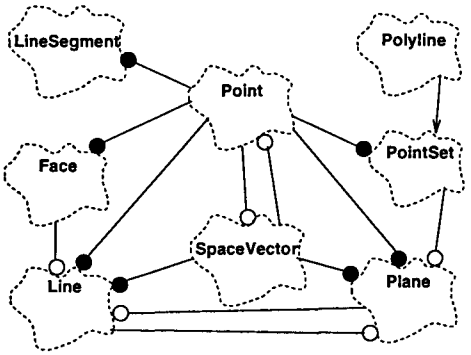

Figure 7: Class diagram of **Geometry** Classes

There are a number of member functions of the Point class. Some of the important ones are:

- *Distance*: finds distance of a Point object from another.
- *Transform*: operates on a Point object and transforms its location based on a transformation matrix.
- *operator* +: transforms a Point object by a vector.
- *AlphaPoint*: finds the location of a Point between two Point objects.
- *operator* −: operates on two Points to give a direction vector.
- *operator* ==: checks if two Point objects are the same.

The PointSet class can contain upto N instances of Points. The SpaceVector class has attributes similar to the Point class. It uses objects of Point class.

The Line class is a composite class (i.e. it contains instances of other objects). Its instances contain exactly one Point and one SpaceVector object. The Point object represents the point through which the line passes and the SpaceVector object gives the direction of the line. Some of the member functions of the Line class are given in the Table 1. Two Line objects *Intersect* to give a Point object. The member function *IsPointOnLine* checks if a Point object lies on a Line.

The Plane class is also a composite class. Its instances contain a Point and a SpaceVector. The Point object lies in the Plane and the SpaceVector is the normal to the plane. Some of the member functions of this class and the tasks achieved by them are outlined in Table 2.

Location of gridnodes. A gridnode is completely defined by its coordinate locations, i.e., a point in 3D. The gridnodes form a set of points. The location of these points can be achieved in many ways. One way is to use a geological package to output the node locations which describe the reservoir geometry. FLEX can use these node locations to construct gridblock geometry. Another way is to use geometrical modules as discussed by Palagi and Aziz (1992) and Verma and Aziz (1996). This simple approach is also implemented in FLEX.

The geometrical modules define point locations on some simple geometries. These point locations can be scaled, rotated, translated and then added to the set of already existing nodes. If there are existing nodes inside the region of a new module to be added then they must be removed. A 3D Cartesian module is a simple example. Such a module can be typically described by the number of points in x, y and z directions and their spacing. The class diagrams in Figures 8 and 9 capture some of the design decisions regarding geometrical modules.

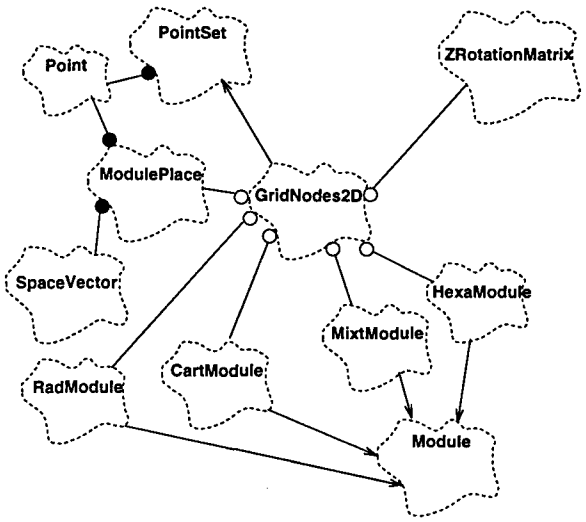

Figure 8: Class diagram of two-dimensional modules

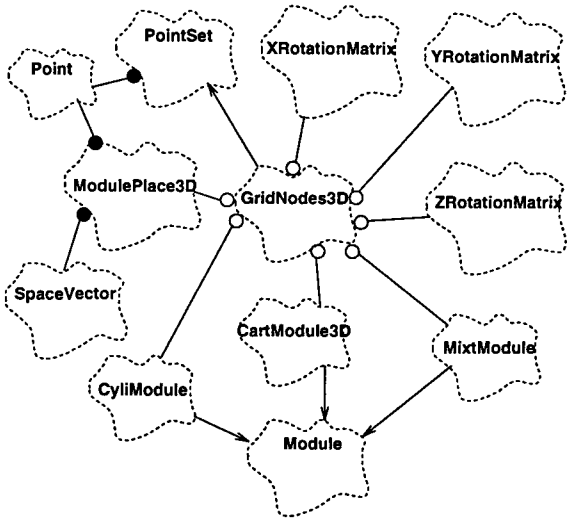

Figure 9: Class diagram of three-dimensional module classes

All modules are ultimately instances of a generalized abstract class named Module. The Module class encompasses the behavior common to all modules and has some pure virtual functions. Some of the two-dimensional module classes derived from the Module class are shown in Figure 8. RadModule class has points located on concentric circles, Hexamodule

Table 1: Some Member Functions of Line Class

Return Type	Member Function	Arguments	Objective
Point	Intersection	Line &B	Intersection of two lines
int	IsPointOnLine	Point &A	Check if point A on line
int	AreLinesCoplanar	Line &B	Are two lines coplanar
int	AreLinesParallel	Line &B	Check if two lines parallel
int	DoLinesIntersect	Line &B	Do the two lines intersect
int	ArePointsSameSide	Point &P1, Point &P2	Are points on same side
Line	Bisector	Line &L1	Find line which bisects 2 lines

Table 2: Some Member Functions of Plane Class

Return Type	Member Function	Arguments	Objective
Line	Intersection	Plane &P2	Intersection of two planes
int	AreParallel	Plane &P2	Check if planes parallel
int	DoPlanesIntersect	Plane &P2, Plane &P3	Do three planes intersect
Point	Intersection	Plane &P2, Plane &P3	Intersection of three planes
Point	Intersection	Line &L	Intersection of plane with line
int	IsPointInPlane	Point &P1	Check if point is in plane
int	IsPointInside	Point &inside, Point &P	Check if P inside half plane
int	DoesLineIntersect	Line &L	Does line L intersect plane
Vector	OutwardNormal	Point &inside	Outward normal

has points located to give hexagonal Voronoi grids in two-dimensions, and MixtModule is just a set of points. All these modules are used by the GridNodes2D class to locate the gridnodes in two - dimensions. GridNodes2D is derived from the PointSet class. The attributes of GridNodes2D are the same as PointSet class. It has a few more member functions, e.g., a function called *AddModule* which can place a module using the attributes of ModulePlace. The ModulePlace class contains instances of Point and SpaceVector classes.

In three-dimensions two modules are defined. They are CartModule3D and CyliModule3D. These two modules are used by the member functions of GridNodes3D and placed using the attributes of ModulePlace3D.

The classes in the **GridNode** subsystem are summarized below:

- Module: Abstract base class having some common functions defined.
- GridNodes2D: The gridnode locations in two-dimensions.
- ModulePlace: Stores information on where and how to place two-dimensional modules.
- CartModule: Derived from Module class. Produces gridnode locations in two-dimensions on a Cartesian grid.
- RadModule: Derived from Module class. Produces gridnode locations in two-dimensions on concentric circles.
- HexaModule: Derived from Module class. Produces gridnode locations in two-dimensions for hexagonal Voronoi grid.
- HexaTiltModule: Derived from Module class. Produces gridnode locations in two-dimensions for tilted hexagonal grid.
- MixtModule: Derived from Module class. Has an arbitrary set of points.
- CartParModule: Derived from Module class. Produces gridnode locations in two-dimensions on a parallel Cartesian grid.
- GridNodes3D: The gridnode locations in three-dimensions.
- ModulePlace3D: Stores information on where and how to place modules in three-dimensions.
- CyliModule3D: Derived from Module class. Produces node locations in three-dimensions on concentric cylinders.
- CartModule3D: Derived from Module class. Produces gridnode locations in three-dimensions on a Cartesian grid.

Connectivity Classes. There are a number of classes defined for grid connectivity information. This connectivity information is useful in the construction of grid block geometry. Given the location of nodes in 3D, a Delaunay triangulation of the nodes will give a tetrahedral decomposition of the reservoir domain. There are four fundamental geometrical elements in the system. They are a set of tetrahedra, a set of triangular faces, a set of edges and, a set of nodes. Each of the entries is unique in the sets. Each geometrical entity can be given an identification number (ID). The ID specifies the location of the

entity in the set.

The Tetra class represents a tetrahedron object. A tetrahedron has four vertices. Its vertices are represented by an integer array of size 4. These integers actually represent the location of the vertex point in a GridNodes3D object, the locations of the gridnodes. A tetrahedron has four triangular faces, each triangle has three edges and each edge has two nodes at its ends. So TetraFace, TetraEdge and TetraNode classes have been defined. The list of these geometrical entities is maintained in their respective sets. Those sets are TetraSet, TetraFaceSet, TetraEdgeSet and TetraNodeSet classes.

Other than providing connectivity information these classes also have member functions which can be invoked to provide many geometrical properties of these objects. Some of these properties are volume of a Tetra object, area of a TetraFace object, length of a TetraEdge object, etc. There are a number of other member functions of these classes which allow other classes to interact with them, e.g., one of the functions determines whether a given Point object lies inside a Tetra object.

The relationship between the **Connectivity** classes is given in Figure 10. These set of classes only have "has" and "using" relationships.

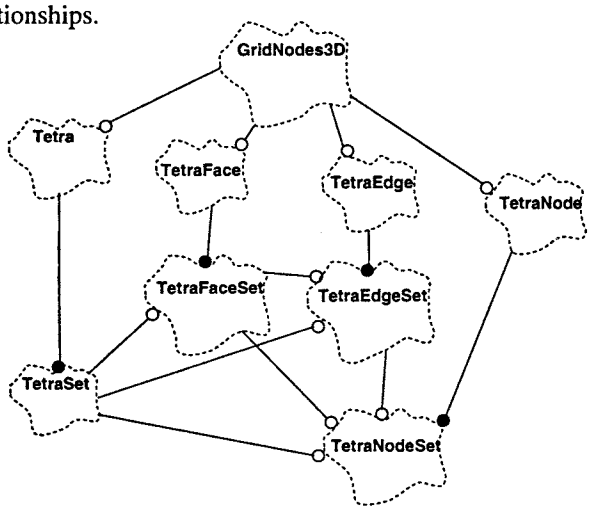

Figure 10: Class diagram of **Connectivity** classes

Grids and Related Classes. A grid is a collection of gridblocks. A gridblock has an associated grid node, a number of faces and has some volume. FLEX uses several types of grids. It also uses connections to construct the Jacobian matrix and residual vector. The properties of a connection are dependent on the grid geometry and the permeability associated with the gridnodes.

Flexible grids are primarily of two types: two- and three-dimensional. A third kind which uses stacks of two-dimensional grids in the third dimension is also defined and is called $2\frac{1}{2}$D grid. These grids can be point-distributed or block centered (see Aziz and Settari, 1979). All the flexible grids in FLEX are point-distributed in both two- and three-dimensions. They can be block-centered in the third dimension for the $2\frac{1}{2}$D grids. The Cartesian grids can however be point-distributed or block-centered in any of the three dimensions.

Based on above analysis, a Grid class is defined which has most of the common functionality of all the grids, e.g., grid visualization and function to create a set of connections. It is made an abstract base class. Default implementation of a function to draw grid geometry is provided in Grid class. This function, called *MathematicaPlot*, is declared to be a virtual function. There is no default implementation for the *CreateConnectioSet* function. It is declared to be a pure virtual function. This function has to be defined for each derived grid class.

The different types of grids are divided into four groups: Voronoi, CVFE and BAG, streamline, and Cartesian grids. They use the Grid class and derive from it. This relationship is shown in Figure11.

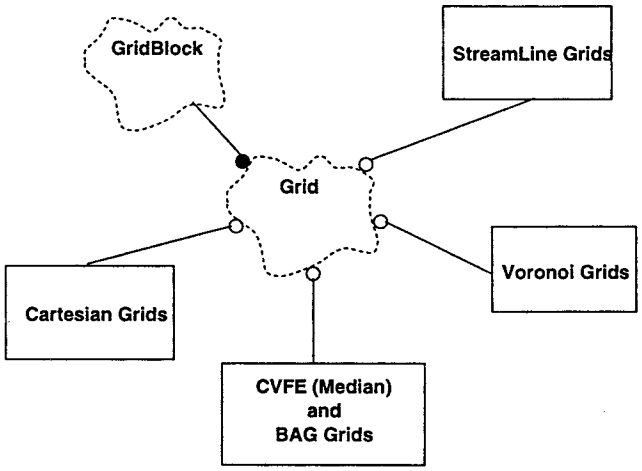

Figure 11: The set of **Grid** classes

Cartesian and Streamline Grids Figure 12 shows the class diagram of the Cartesian and streamline grids. Most the classes are derived directly from the Grid class. The **WhiteGridPDXY** grid class is derived from CartGridPDXY class. The *CreateConnectionSet* member function is redefined for each of these grids. The main features of these grid classes are outlined below:

- CartGrid: This is a block-centered Cartesian grid. It creates ConnectionSet object to simulate flow in one-, two- or three-dimensions, which gives three-, five- or seven-point formulation, respectively.
- CartGridPD: This is a point-distributed Cartesian grid. It creates ConnectionSet object to simulate flow in one-, two- or three-dimensions, which gives three-, five- or seven-point formulation, respectively.
- CartGridPDXY: This is a point-distributed grid in x-y plane and block-centered in the z direction.
- ShirGridPDXY: This is a point-distributed grid in x-y plane and block-centered in the z direction. It derives the

gridblock geometry functionality from CartGridPDXY class. The ConnectionSet object it creates gives a nine-point formulation in x-y plane. The flow along all the connections is a function of potentials at the nodes of this connection only, i.e., only two nodes. This formulation gives a nine-point formulation even for diagonal permeability tensors. In three-dimensions an eleven-point formulation results.

- WhiteGridPDXY: This is a point-distributed grid in x-y plane and block-centered in the z direction. The flow along its x-y connections is a function of six potentials. It gives a nine-point formulation for full permeability tensors. In three-dimensions it gives an eleven-point formulation. For diagonal permeability tensors it gives five-point formulation in two-dimensions and seven-point formulation in three-dimensions.
- StreamLineGrid: This grid creates gridblocks which have their boundaries aligned along streamlines. It is defined for two-dimensional flow only. The connections created by this grid give a nine-point formulation, even for diagonal tensors.

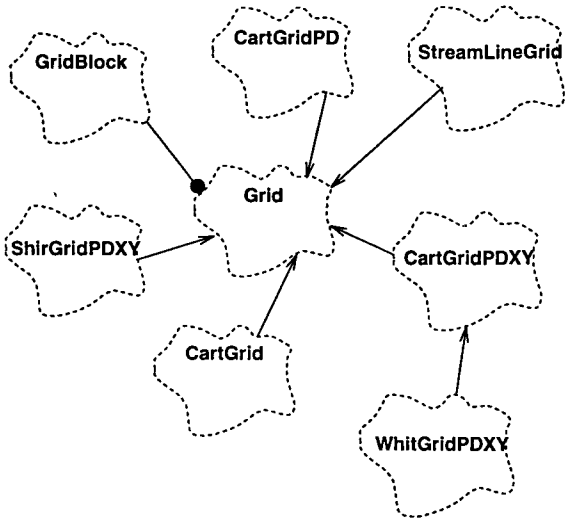

Figure 12: Class diagram of Cartesian grids

Voronoi Grids. Figure 13 shows the class diagram of Voronoi grids implemented in FLEX. VoronoiGrid2D and VoronoiGrid3D derive from the Grid class. The main features of these grids are outline below:

- VoronoiGrid2D: This is a very general implementation of Voronoi grids in two-dimensions.
- VoronoiGridSimple2D: This creates the Voronoi grids with certain restrictions in the gridblock geometry (see Verma and Aziz, 1996) which allows it to be used for flow calculations with full tensor permeability distribution.
- VoronoiGridSimple2HD: This is an extension of VoronoiGridSimple2D to model vertical variations in reservoir properties and uses stacks of two-dimensional simple Voronoi grids.
- VoronoiGrid3D: This is a very general implementation of Voronoi grids in 3D. The *CreateConnectionSet* member function for this grid and flow calculation on this grid has not yet been implemented in FLEX.

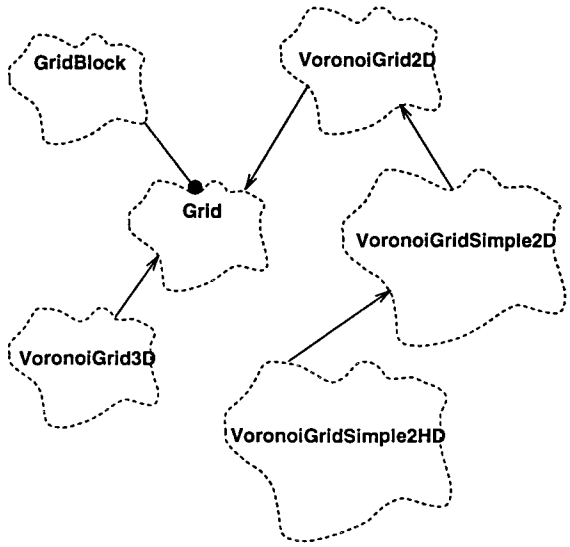

Figure 13: Class diagram of Voronoi grids

Figure 14 shows the class diagram of CVFE (i.e. median) grids and its variations implemented in FLEX. A maximum of four levels of inheritance is used in the class relationships. Multiple inheritance is used in one instance. Thus, CvfeGrid2HD_BC inherits from both CvfeGrid2D and Grid2HD_BC. Both these classes are derived from the same base class, i.e., Grid class. It order to avoid dual instances of data members of Grid class being instantiated, Grid class has to be defined as a virtual base class in both Grid2HD_BC and CvfeGrid2D classes. Some of the important aspects of the definition of CvfeGrid2D and CvfeGrid2HD_BC are shown below:

```
class CvfeGrid2D: public virtual Grid {
protected:
  //Additional data members.
public:
  //A few constructors.
  CvfeGrid2D();
  CvfeGrid2D(int sz, GridBlock *cvgrid);
  CvfeGrid2D(TetraFaceSet & tfset, PointSet & pSet, double thick);
  //More member functions.
};

class CvfeGrid2HD_BC: public Grid2HD_BC, public CvfeGrid2D {
protected:   //Additional data member
public:   //A few constructors.
  CvfeGrid2HD_BC();
  CvfeGrid2HD_BC(int sz, GridBlock *cvgrid);
  CvfeGrid2HD_BC(TetraFaceSet &tfset, PointSet &pSet,
     DArray &layerThick);

  //More member functions.
};
```

Some of the other **Grid** classes shown in the class diagram and their main features are:

- Grid2HD_BC: This creates grid having stacks of two-dimensional grids (i.e. $2\frac{1}{2}$D grids).
- CvfeConstPermGrid2D: This creates CVFE grid with constant permeability fundamental elements.
- CvfeVorBagGrid2D: This creates grid with CVFE criterion, then modifies the gridblock boundaries to satisfy Voronoi criteria as much as possible and finally modifes the boundaries to align with layer boundaries.
- CvfeBagGrid2D: This creates grid with CVFE criterion and then modifes the boundaries to align with layer boundaries.
- CvfeGridXZ: This crteatse CVFE grid in XZ plane.
- CvfeGrid3D: This creates three-dimensional CVFE grid.
- CvfeConstPermGrid3D: This cretase three-dimensional CVFE grid with constant permeability fundamental elements.
- CvfeBagGrid3D: This creates CVFE-BAG grids in three-dimensions.
- CvfeBagConstPermGrid3D: This creates CVFE-BAG grid with constant permeability fundamental elements.
- GpebiGrid2D: This creates generalized perpendicular grids (GPEBI) in two-dimensions.
- GpebiGrid2HD_BC: This creates $2\frac{1}{2}$D GPEBI grids.

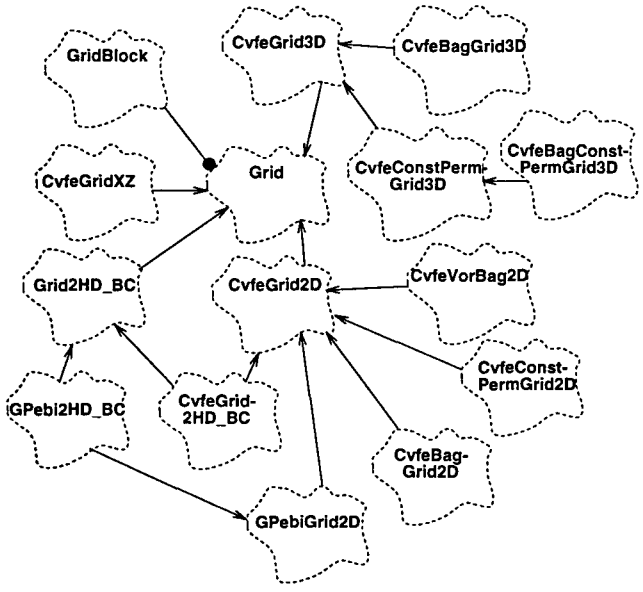

Figure 14: Class diagram of CVFE and other grids

The constructors of these grid class use the PointSet class and connectivity information to generate the geometry of the grid blocks. For the Voronoi grid and CVFE grid generation the PointSet and connectivity information is sufficient to generate the geometry of gridblocks. The CVFE-Voronoi-BAG grid needs additional information to align grid block faces along known reservoir boundaries. In two-dimensions, these boundaries are implemented in the Polyline class. In three-dimensions, boundaries in the form of surfaces have to be specified.

Reservoir Classes. The class relationships for this subsystem is given in Figure 15. The classes in this set colloborate to compute flow between the gridnodes due to various boundary conditions and the resulting changes in the state variables over a finite time interval. A Reservoir class has a number of ControlVolumes (one associated with each gridnode in the reservoir domain). The ControlVolume is at a certain State and has some Oil and and Water Reserves. The State of the ControlVolume is defined by pressure and phase saturations. The Reserves in the ControlVolumes make up the total existing Reserves of the Reservoir. Flow between ControlVolumes is through Connections. In two-dimensions, flow along a Connection can be divided into flow in two triangles. Such a flow is defined as a SubConnection flow. The connection approach enables the same flow simulation code to be used with any control-volume type numerical method, e.g., Cartesian, Voronoi, Generalized PEBI grid, CVFE and Mixed Voronoi-CVFE grid. A ConnectionSet class is constructed based on the type of grids used.

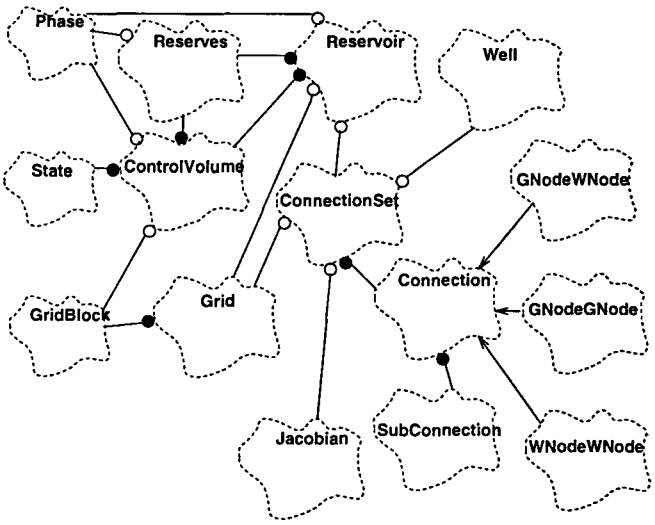

Figure 15: Class diagram of **Reservoir** classes

Different kinds of connections are possible. Flow can be between two gridnodes, between a wellnode and a gridnode, between two wellnodes, etc. All such flows can be handled in a similar manner. The main advantage in dealing with connections is the ease with which any control volume formulation can be programmed. Residual and Jacobian can be very conveniently constructed using the concept of connections. Wells are also very conveniently handled using this concept.

Conclusions

This paper presented a detailed object-oriented analysis and design of FLEX, an object-oriented reservoir simulator. Several flexible grids were implemented in FLEX. Through the mechanism of inheritance, it was possible to design code that allowed easy extensions to different grid geometries.

Inheritance allowed reusing tested and dependable code and thus it was possible to have relatively bug-free extensions of the code.

Acknowledgements

The authors acknowledge the financial support of the Horizontal Well and Supri-B Industrial Affiliates programs of Stanford University. A part of this work was also supported by DOE contract No. DE-FG22-93BC14862.

References

1. Aziz, K. and Settari, A.: *Petroleum Reservoir Simulation*, Applied Science Publishers Ltd., London (1979).
2. Cheriton, D.R.: *Object-Oriented Programming from a Modeling and Simulation Perspective*, Course Material for CS 249, Stanford University, Winter 1996, to be published by McGraw-Hill.
3. Palagi, C.L. and Aziz, K.: "Use of Voronoi Grids in Reservoir Simulation," paper SPE 22889 presented at the 66th Annual Technical Conference and Exhibition, Dallas, October 6-9,1991.
4. Lim, K.T., Schiozer, D.J., and Aziz, K.: "A New Approach for Residual and Jacobian Array Construction in Reservoir Simulators," SPE Computer Applications, Volume 7, Number 4, August 1995, 93-97.
5. Lippman, S.B.: *C++ Primer*, Addison Wesley, 2nd edition (1991).
6. Booch, G.: *Object-Oriented Analysis and Design with Applications*, The Benjamin/Cummings Publishing Company, Inc., second edition (1994).
7. Verma, S.K., and Aziz, K.: "Two- and Three-Dimensional Flexible Grids in Reservoir Simulation," paper presented at the 5th European Conference on the Mathematics of Oil Recovery, Leoben, Austria, 3-6 Sept, 1996.