International Conference on Mathematics, Computational Methods & Reactor Physics (M&C 2009)
Saratoga Springs, New York, May 3-7, 2009, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2009)

EFFECTIVE SOFTWARE DESIGN FOR A DETERMINISTIC TRANSPORT SYSTEM

A. M. Watson, R. E. Grove, and M. T. Shearer
Knolls Atomic Power Laboratory
P.O. Box 1072, Schenectady, NY 12301
watsam@kapl.gov, grovero@kapl.gov, and shearer@kapl.gov

ABSTRACT

Jaguar is a new deterministic transport-based tool that is being developed for reactor physics and shielding applications. The Jaguar development effort is using object-oriented design (OOD) principles in the design of Jaguar. First, we briefly discuss the requirements and capabilities of Jaguar. We then describe how OOD is used in the design of Jaguar.

Key Words: Software Engineering, Jaguar, Discrete Ordinates, SBA

1. INTRODUCTION

Jaguar is a deterministic transport-based design tool being developed for reactor physics and shielding design applications. It is being written in C++ and Fortran, and uses modern object-oriented design and programming constructs to increase flexibility, maintainability, and extensibility. Jaguar currently uses the Slice Balance Approach (SBA) to solve the discrete ordinates equations on arbitrary polyhedral meshes, and will include a polyhedral mesh diffusion solver for synthetic acceleration. Other solvers, such as P_N, SP_N, quasi-diffusion, and corner balance, can be added without architectural changes. The SBA is a solver framework for S_N spatial solvers. Currently, Jaguar uses a diamond-difference and a step characteristics scheme, but other methods can be added, including linear characteristics, linear discontinuous, and arbitrarily high-order transport methods (AHOT) [1]. In this paper, we describe the software requirements for Jaguar and show examples of the use of object-oriented design in the design of Jaguar.

2. JAGUAR OVERVIEW

Jaguar is being developed for a range of reactor physics and shielding applications. This application space dictates that Jaguar must be robust enough to handle the different requirements of these two application areas. For some reactor physics applications, users expect accuracy on a fine but topologically simple computational mesh with many material definitions. In contrast, for some shielding applications, the computational mesh is coarse but complicated with few material definitions. Jaguar must be capable of handling both of these extremes of application space efficiently. In addition, it must be capable of solving fixed-source, multiplication eigenvalue, time-absorption eigenvalue, and multiplying source problems, as well as problems with varying degrees of scattering. Jaguar is intended to span the design space from fast scoping calculations that take seconds or minutes to run, to benchmark calculations that may take hours or days to run, trading execution time for accuracy. Jaguar must be flexible enough to run both small problems (a handful of energy groups, dozens of angles, and a few thousand mesh cells) and very large problems (hundreds of energy groups, thousands of angles, and millions of mesh cells) on modern supercomputers. Due to the complexity and scope of these requirements, the Jaguar development team uses object-oriented design principles to address the many issues that result from these extensive requirements.

3. THE DESIGN OF JAGUAR

Jaguar is designed using object-oriented design techniques [2], and is written using C++ and Fortran. The developers leverage the use of Design Patterns [3] and the Standard Template Library (STL) [4] as much as is reasonable for a high-performance computational system. Each of the subsystems in Jaguar contains a group of classes that perform some common set of functionalities, and/or operate on a common set of data. These classes each have a specified interface that allows the user to assemble them to perform some set of operations, which can be general (such as mesh) or specific to radiation transport (such as fission). Jaguar itself is a large library of these subsystems, with a driver program that executes the subsystems in a sequence to solve the radiation transport problem.

3.1. Trilinos

Jaguar uses the Trilinos computational framework extensively. Trilinos [5] is a collection of mathematical software packages designed for use in large-scale engineering and scientific applications. Trilinos uses MPI for parallel computations and for data distribution. The foundation of Trilinos is the Epetra package, which provides the basic classes for defining distributed matrices and vectors, linear operators, and linear problems. All of the Trilinos packages use Epetra objects for input and output. In addition to Epetra, Trilinos includes Krylov-based linear solvers (AztecOO and Belos), Krylov-based eigenvalue problem solvers (Anasazi), non-linear solvers (NOX), multi-level and domain decomposition algorithms (ML), and interfaces to BLAS and LAPACK libraries (Teuchos).

3.2. Subsystems

Jaguar is composed of subsystems, each containing a group of classes that perform a common set of functionalities, and/or operate on a common set of data. The architecture of Jaguar is shown in Figure 1 as a UML Component Diagram. Each of the icons shaped like the icon labeled “GUI Model Builder” is called a package, and represents a component used by the Jaguar system. The icons shaped like the icon labeled “Utilities” are called subsystems, and represent pieces of the Jaguar system. The dashed arrows are called dependency arrows, and denote dependencies of components on other components or packages. The arrows originate at the dependent subsystem and terminate at the component on which the dependent subsystem depends.

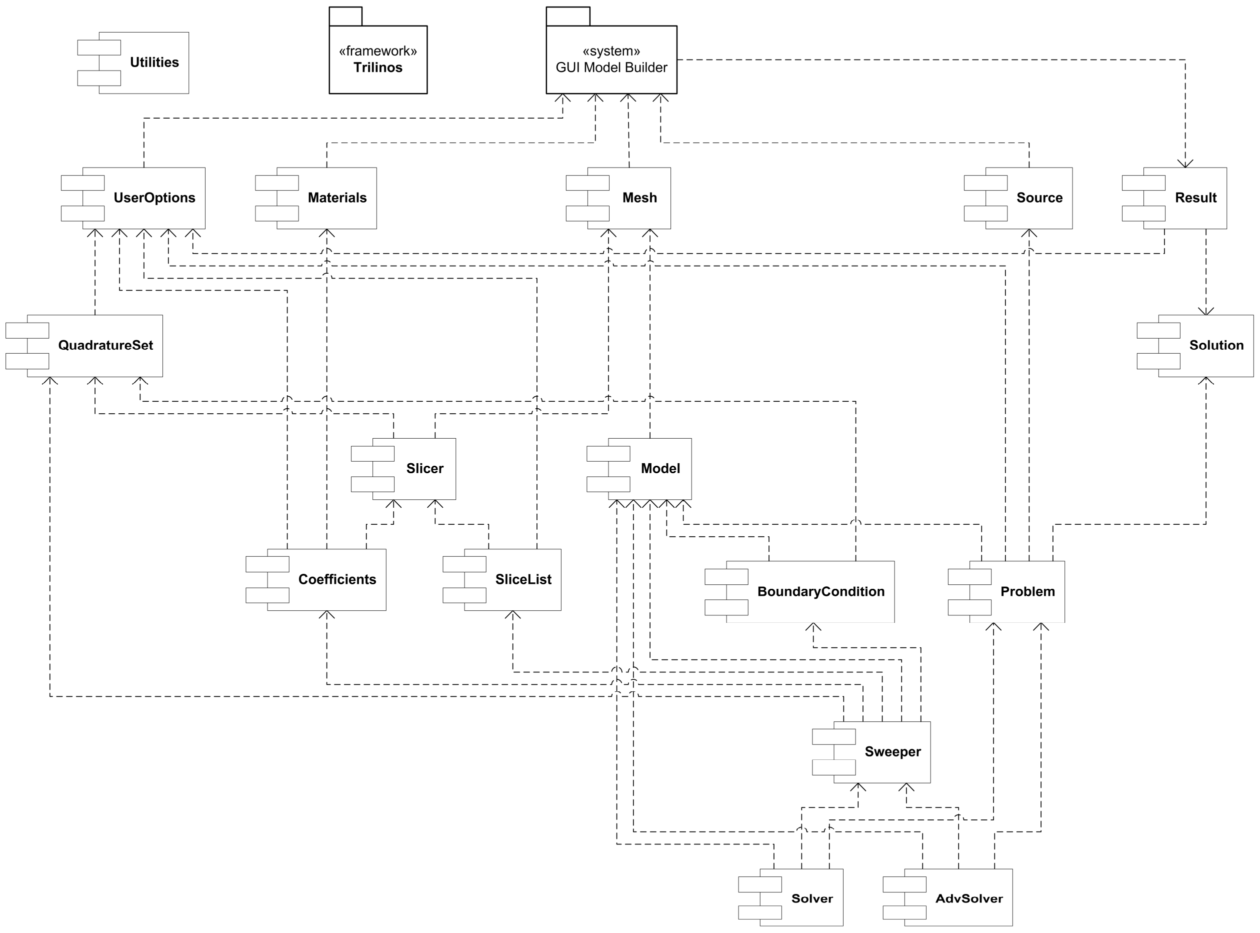

Figure 1. Jaguar architecture component diagram.

The primary interface for Jaguar is with the GUI model builder system through interface files. Each of these interface files requires a file handler to read and/or write the file, and to store and control access to the data contained within the file. At the top of the diagram are the GUI model builder, the Trilinos framework, and the Utilities subsystem. Links between Trilinos and Utilities and other subsystems are not explicitly shown in the figure, as they are so numerous. Immediately below these three items are five subsystems that contain the classes responsible for interfacing with the GUI model builder. These subsystems are labeled UserOptions, Materials, Mesh, Source, and Result in the diagram. The first four subsystems read input from the GUI model builder, while Result writes output to be read by the GUI model builder. Below this first level of subsystems lie the other subsystems that control generating the solution of the specified problem. The QuadratureSet subsystem controls generation of the quadrature set. The Slicer, Coefficients, and SliceList subsystems control SBA-specific features of the solver, and the Problem subsystems actually solves the transport problem itself. Two of these subsystems, Mesh and QuadratureSet, will be explained in further detail in this section.

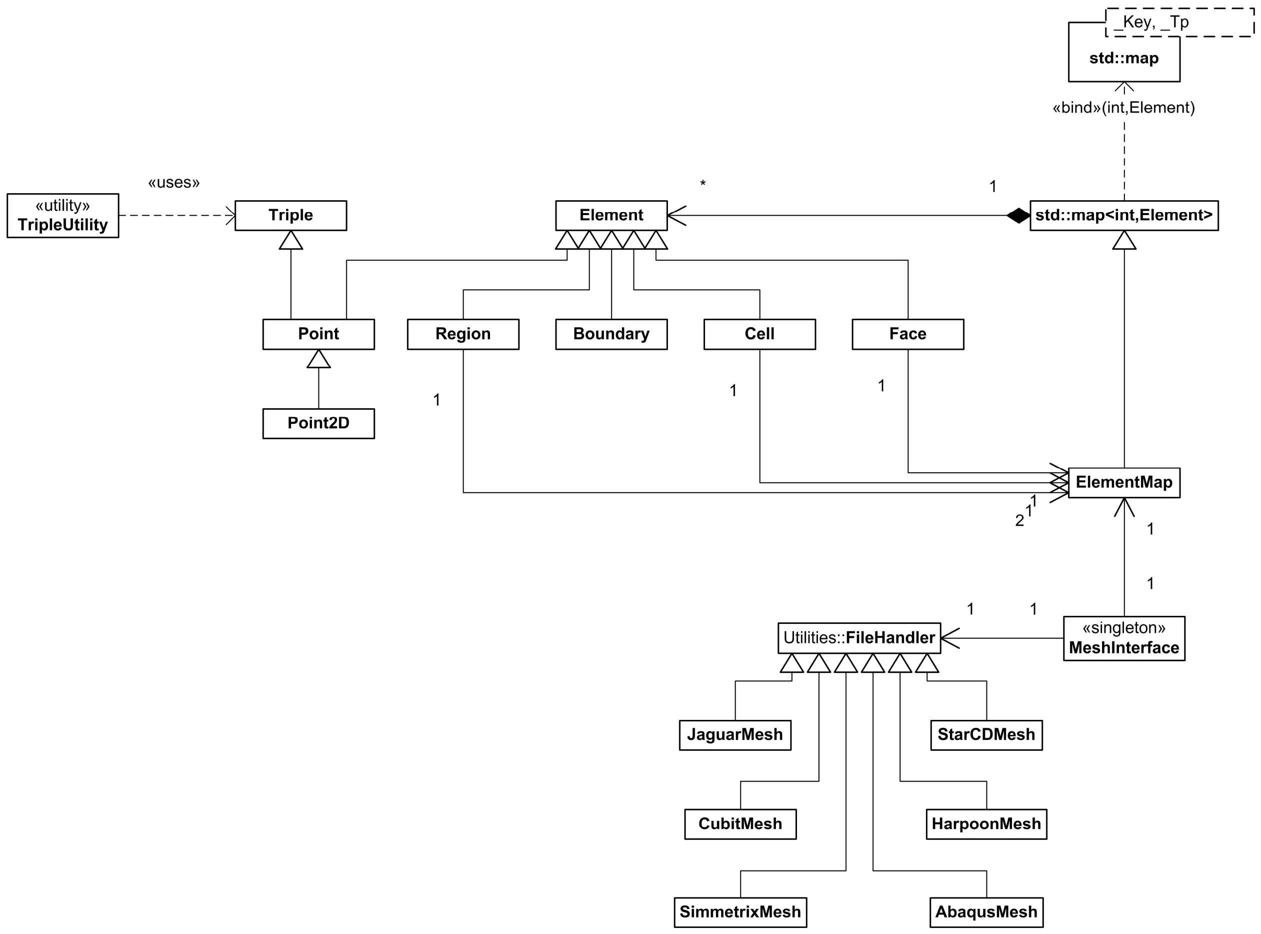

Figure 2. Mesh subsystem high-level class diagram.

3.2.1 Mesh Subsystem

Jaguar uses a non-manifold, 3D arbitrary polyhedral meshed geometry description internal to the code. The Mesh subsystem contains classes that store the computational mesh and perform operations on the mesh. A high-level class diagram of the Mesh subsystem is shown in Figure 2. The data in the subsystem is accessed through an interface class named MeshInterface. This class follows both the *Façade* and *Singleton* design patterns. The MeshInterface class contains a Utilities::FileHandler object, which serves as a base class for the classes that handle the mesh input files, and an ElementMap data structure storing a Region class instance for each quadrature set region in the model.

Several classes derived from Utilities::FileHandler are shown in the figure. These classes are used for handling different types of meshed geometry files. These files may be ASCII or binary, and may use a vendor-provided API for reading and writing the binary formats. The JaguarMesh class handles a file that defines the meshed geometry using a native format. The CubitMesh, SimmetrixMesh, StarCDMesh, HarpoonMesh, and AbaqusMesh classes handle meshed geometry files generated by several commercially available mesh generators. Since these file handlers are all derived from a common interface, additional handlers may be added as necessary.

All of the geometric objects that define the meshed geometry are derived from the Element class. The Element class simply defines that a derived class has an integer identification number. The ElementMap class defines a mapping between this identification number and the memory address of the element itself, and is required for quickly finding the element of interest given only its identification number.

The Triple class defines an object storing a vertex position, represented by three spatial coordinates. This class and the TripleUtility utility class define operations on these vertices. The Point class is used to represent the vertices in the model, and is derived from both the Triple class and the Element class. That is, a Point is a Triple, and a Point is an Element, or, more simply, a Point is a Triple with an identification number. Likewise, Point2D is derived from Point, and represents a two dimensional point. This is represented internally as a Point object with the value of the z-coordinate set equal to zero and inaccessible.

All of the other entities in a model are derived from the Element class. The four primary elements are the Region, the Boundary, the Cell, and the Face. The Region class is derived from Element, and represents a unique quadrature set region. This class stores two ElementMap objects, storing a map of the boundaries of the region and a map of the cells that make up the region. The Boundary class is derived from Element, and represents the boundaries of a region. This class stores the integer identification numbers of the faces that lie on the boundary. The Cell class is derived from Element, and represents a single computational cell. This class stores a map of the faces that make up the cell. The Face class is derived from Element, and represents a planar polygonal face defined by a set of points. This class stores a map of the points that make up the face.

3.2.2 QuadratureSet Subsystem

The high-level class diagram of the QuadratureSet subsystem is shown in Figure 3. The data in the subsystem is accessed through an interface class also named QuadratureSet. This class follows the *Façade* design pattern. The QuadratureSet class contains a Utilities::FileHandler object, which serves as a base class for the classes that handle the user-defined quadrature set input files, an interface to the Epetra_SerialDenseMatrix class from the Trilinos framework, and a QuadratureSetGenerator object for generating standard quadrature sets. In addition, QuadratureSet is derived from an STL vector that stores an Angle class instance for each discrete angle in the quadrature set. The Angle class stores the components and weight for one angle.

The Epetra_SerialDenseMatrix class defines operations for storing and interfacing with small dense matrices that are meant to be stored on one computational node. This matrix is used to store the discrete-to-moments transformation matrix. The XmlFileHandler class, which is shown in the diagram as being derived from the Utilities::FileHandler class, handles a file that defines the quadrature set in an XML format, and interfaces with the Teuchos::XMLObject class for reading and writing the file. Since this file handler is derived from a common interface, additional handlers may be added as necessary.

Several classes are shown in the figure using the QuadratureSetGenerator as a base class. These classes are used for generating different standard quadrature sets. The GauLegGenerator class

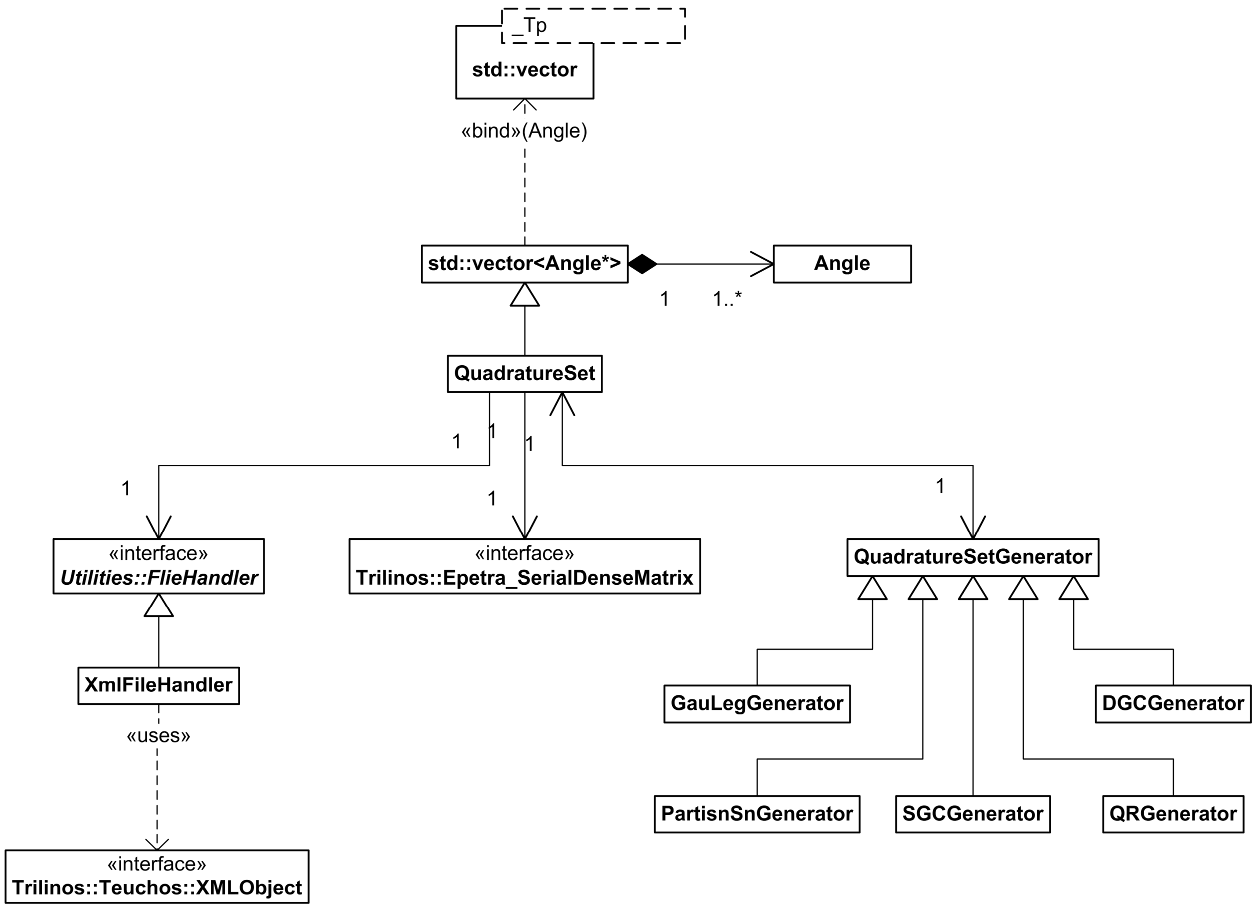

Figure 3. QuadratureSet subsystem high-level class diagram.

generates a completely symmetric quadrature set satisfying even moment conditions using LA-3186 [6] as the primary reference. This is identical to the level symmetric quadrature set used in Attila, and is the default quadrature set type in Jaguar. The PartisnSnGenerator class generates a quadrature set that is equivalent to the default quadrature set used in PARTISN [7]. The SGCGenerator and DGCGenerator classes generate single and double Gauss-Chebyshev quadrature sets, respectively. These can have rectangular, triangular, or user-defined arrangements. Finally, the QRGenerator class generates the quadruple range quadrature sets [8].

3.3. Operators

Jaguar supports four primary problem types: fixed-source problems, multiplication eigenvalue problems, time-absorption eigenvalue problems, and multiplying source problems. In order to take advantage of the flexibility offered by object-oriented design and the Trilinos framework, these problem types use a common operator base class named Operator. The transport operators in Jaguar are all derived from Operator, and Operator in turn is derived from the Epetra_Operator class provided by Trilinos. The linear solvers in AztecOO are written to use classes derived from the Epetra_Operator class for both the primary operator and the preconditioner operator for a linear problem. Since all of Jaguar's operators are derived from this class, any combination of them can be used to solve a particular problem. This means that, for example, the DiffusionOp-

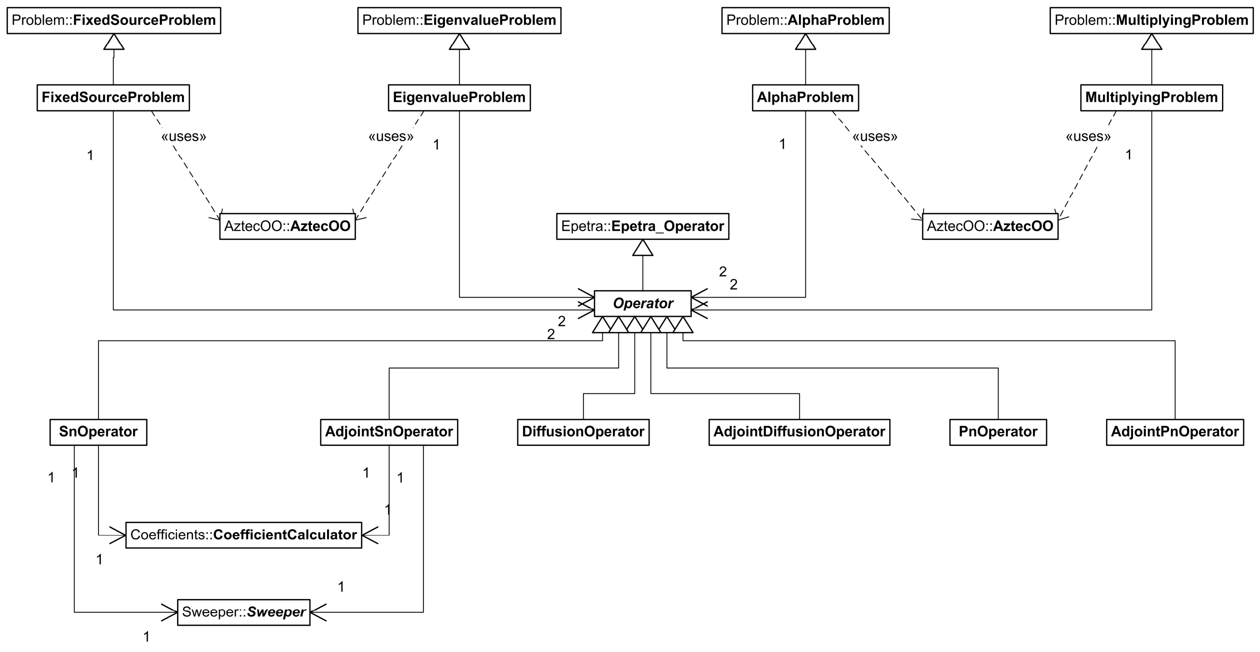

Figure 4. Solver subsystem high-level class diagram.

erator class can be used as the primary operator for solving a diffusion problem, or as a preconditioner operator for acceleration using DSA, without any changes to the operator. The initial operators in Jaguar are the SnOperator, which uses the SBA to solve the discrete ordinates equation on polyhedral meshes, and the DiffusionOperator, which solves the diffusion equation on polyhedral meshes. Additional operators that may be added in the future include a P_N operator, a Simplified-P_N operator, and a quasi-diffusion operator, as well as adjoint operators for each of the corresponding forward operators. All of these operators are part of the Solver subsystem.

3.3.1 Solver Subsystem

The high-level class diagram of the Solver subsystem is shown in Figure 4. This subsystem contains the routines that solve each of the respective problem types and the operators that define the action of the multigroup transport operator on a solution vector. Each of the operator classes is derived from the Operator class, which is in turn derived from the Trilinos Epetra_Operator class. The SnOperator and AdjointSnOperator classes provide the SBA-based multigroup discrete ordinates forward and adjoint operators, respectively, and interface with the Coefficients:: CoefficientCalculator class for computing the SBA coefficients and the Sweeper::Sweeper class for performing the transport sweep. The DiffusionOperator and AdjointDiffusionOperator classes provide multigroup forward and adjoint diffusion operators, respectively. The PnOperator and AdjointPnOperator classes provide multigroup forward and adjoint operators, respectively, that use a spherical harmonics approximation in angle. Each of these operators can be used either as the primary solution operator or as a preconditioner operator.

The FixedSourceProblem class contains the interfaces that provide for the solution of a fixed source problem, and is derived from the Problem::FixedSourceProblem class. In particular, the class contains two Solver objects, the primary solver that defines the transport operator and the preconditioner solver that defines the desired preconditioner operator. The class also interfaces with the Trilinos linear system solvers through the AztecOO package. This package defines the various linear solvers that will be used for the energy group iteration (i.e. GMRES, BiCGStab, etc.). The EigenvalueProblem class contains the interfaces that provide for the solution of a multiplication (*k*) eigenvalue problem, and is derived from the Problem::EigenvalueProblem class. This class also contains two Solver objects for the primary solver and the preconditioner, as well as a FissionOperator object that defines the fission operator. The class also interfaces with the Trilinos linear system solvers through the AztecOO package and the Trilinos eigenvalue solvers through the Anasazi package. The Anasazi package defines the Krylov eigenvalue solvers that are used for the eigenvalue iteration.

4. CONCLUSIONS

We have described a high- and mid-level view of the design of Jaguar. Since Jaguar is a new development effort, it is much easier to explore new software design philosophies than when maintaining or extending an existing system. We have found that using the principles of object-oriented design help us to create a flexible tool, and we expect that this approach will also make it easier to maintain and extend Jaguar in the future. Our experience to date working with new developers and adding new functionalities gives us confidence in this expectation. For example, new developers have implemented different quadrature sets and different coefficient calculators in Jaguar with no impact on the rest of the code. We have also had developers extend the capabilities of several of the components of the Slicer independent of other development efforts. These changes have been incorporated seamlessly.

In this paper, we have described the software requirements for Jaguar, given an overview of the principles of object-oriented design, and shown examples of the use of object-oriented design in the design of Jaguar. In doing so, we have illustrated one application of object-oriented design to the development of a deterministic transport tool.

REFERENCES

1. Y.Y. Azmy, “Arbitrarily High Order Characteristic Methods for Solving the Neutron Transport Equation,” *Ann. Nucl. Energy*, **19**, 593–606 (1992).
2. G. Booch, *Object-Oriented Analysis and Design with Applications*, 2nd ed., Addison-Wesley, Reading, Massachusetts, USA (1994).
3. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading Massachusetts, USA (1995).
4. N.M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley, Boston (1999).
5. M. Sala, M. Heroux, and D. Day, “Trilinos Tutorial,” SAND2004–2189, Sandia National Laboratories (2004).
6. K.D. Lathrop and B.G. Carlson, “Discrete Ordinates Angular Quadrature of the Neutron Transport Equation,” LA-3186, Los Alamos Scientific Laboratory (1964).
7. R.E. Alcouffe, R.S. Baker, J.A. Dahl, S.A. Turner, and R.C. Ward, “PARTISN: A Time-Dependent, Parallel Neutral Particle Transport Code System,” LA-UR-05-3925, Los Alamos National Laboratory (2005).

8. I.K. Abu-Shumays, "Angular Quadrature for Improved Transport Computations," *Trans. Theory and Stat. Physics*, **30**:169 (2001).