AIAA 2004-1262

Application of Object-Oriented Design Patterns to a CFD code with an Adaptive Mesh Refinement Technique

H. Katsurayama, K. Komurasaki, and Y. Arakawa
The University of Tokyo
Tokyo, Japan

42nd Aerospace Sciences Meeting & Exhibit
5–8 January 2004
Reno, Nevada

AIAA 04-1262

Application of Object-Oriented Design Patterns to a CFD Code with an Adaptive Mesh Refinement Technique

Hiroshi KATSURAYAMA[*], Kimiya KOMURASAKI[†] and Yoshihiro ARAKAWA[‡]

The University of Tokyo, Hongo 7-3-1, Bunokyo, Tokyo, 113-8656, Japan

ABSTRACT

Object-Oriented design patterns for a general CFD code were suggested, and were also applied to an Adaptive Mesh Refinement(AMR) using unstructured mesh. For a test computation, the shock tuble problem is solved. In addtion, a run-time speed between a procedural and OOP code is compared. The run-time speed of the C++ code is about 7 times as slow as the Fortran code. However, the flexibilty and reusability of the code are dramatically enhanced by OOP.

INTRODUCTION

Until now, most of CFD applications had been coded by FORTRAN language. Although FORTRAN CFD codes are efficient, the codes tend to lose reusability, flexibility and extensibility for solving complex problems, for example, an Adaptive Mesh Refinement (AMR).

Recently, several researchers suggest the applications of Object-Oriented Programming (OOP) to CFD.[1,2] The OOP overcomes the drawbacks of traditional programming fashion by class encapsulation, polymorphism and class inheritance.[3] In OOP, calculations proceed by object-communications. For example, in C++ language, a class produces objects, and they communitcates each other through their operations. Thereby, it is important what classes should be extracted from CFD problems. If the class extractions are poor, the code may be more complex than the procedural programming. Gamma et al.'s "Design patterns"[4] propose the solutions for this problem. Their general design pattern identifies the participating classes and objects, their roles and collaborations, and the distribution of responsibilities.

There are several commercial CFD softwares, they may be coded by OOP fashion. However, most of CFD codes in laboratory level would be coded by traditional fashion, and their mentaince would be quite troublesome. In this paper, OOP design patterns for a general CFD code are suggested, and are also applied to an Adaptive Mesh Refinement(AMR) using unstructured mesh. In addition, a shock tube problem is solved for a test computation.

Finally, run-time speeds of Fortan and C++ code are compared.

NUMERICAL SCHEME

The 2-D compressible Euler equation is solved using unstructured cell-centered mesh. Inviscid flux is calculated by AUSM-DV[6] scheme which is extended to 2nd order spatial accuracy by MUSCL approach with Venkatakrishnan's limiter.[7] For AMR, Rivira's bisection algoritm[8,9] is used.

OBJECT-ORIENTED PROGRAMMING

In order to code by objecct-oriented fashion, three mechanisms of class encapsulation, polymorphism and class inheritance are indispensible. C++ and Java language has the mechanisms, and are famous for the representative OOP language. Though Java recently becomes very popular in a general software development, the run-time is still slow for scientific calculations. The run-time of C++ is not too slow, and it has many useful libraries.[10–13] Thereby, we choose C++ to code a CFD program by OOP.

Class Encapsulation

Figure 1 shows a simplified class diagram of a `Cell` class, using UML(Unified Modeling Language).[5] A box expresses a class. The rows in the box designate the name, data member, and member operations. A minus sign expresses a private member, and a plus sign expresses a public member. The private members are not directly operated from other objects but only by public member operations.

[*]Graduate student, Department of Aeronautics and Astronautics, Student Member AIAA

[‡]Associate Professor, Department of Advanced Energy, Member AIAA

[§]Professor, Department of Aeronautics and Astronautics, Member AIAA

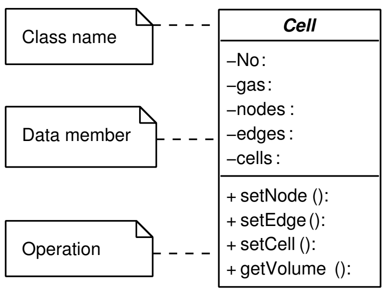


Figure 1: Class Encapsulation.

Since the data related to an object can be bundeld to one place and be operated by own public operations, the code becomes easy to read and mistakes in coding decrease.

Class Inheritance and Polymorphism

Figure 2 shows an example of class inheritance of the `Cell` class. A closed arrow shows a class inheritance, herein, a subclass points its superclass. An italic class name means an abstract class. It defines the common operations of subclasses, and the outlines of diffrent operations between subclasses. Diffrent operations are specilized in each subclass(concrete class).

By the class inheritance, the mechanism of polymorphism can be utilized. If the following loop conducts for the data set of pointer of `Edge` as shown in Fig.3,

```
for ( data set of pointer of Edge)
  data->operation();
```

the specilaized operations for each concrete class is called. By this mechanism, a client calss does not has to know the true type. As the result, the concrete class is encapsulated from its client class. In addition, when the client needs other concrete class, the new concrete classes are only added. Its modification never spread over all the code. This promotes the reusability and extensibility of a code.

APPLICATION OF DESIGN PATTERNS TO A CFD CODE

Data Structure

Figure 4 shows the class hierarchy of mesh components. An abstract `Cell` class are desined to have the concrete classes, `TriangleCell`, `SquareCell` and `GhostCell<int>`. All interior cell objects are created from the `TriangleCell` or `SquareCell`. The `GhostCell<int>` class is defined as a template class. For example, A `GhostCell<0>` object is a ghost cell outside of a wall. Only one object is created from each ghost object and shared by interior cells or boundary edges as shown in Fig.5.

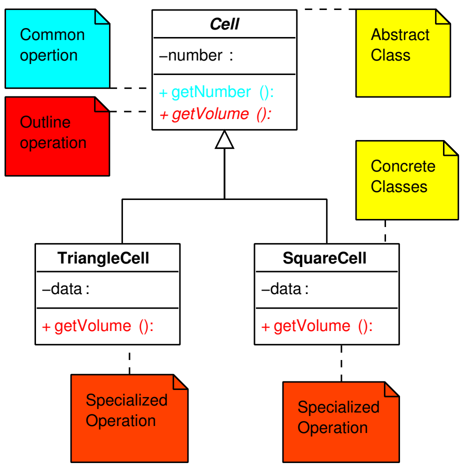


Figure 2: Class Inheritance.

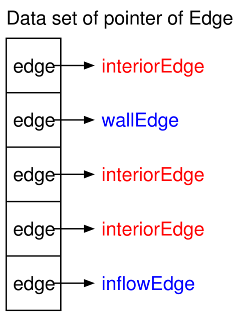


Figure 3: Polymorphism.

An abstract `Edge` class is also designed to have a `InteriorEdge` and `BoundaryEdge<int>` concrete class. Unlike the `GhostCell<int>` class, All edge objects on boundaries are created from the `BoundaryEdge<int>` class.

A `Node` class is designed as a single concrete class. Whether a `Node` object is located on boundaries or not can be judged by refering the `Edge` type belonging to itself.

Relations between mesh components are also shown in Fig.4. A normal line indicates the relations between classes. To create the mesh or to proceed the CFD calculations in the unstructured scheme, a mesh componet has to know other components which construct or surround itself. The `Cell` object has the `vector` containers of pointers of its adjacent cells, consitutive nodes and edges. The `Edge` object has the `vector` containers of pointers of its consitutive nodes and two adjacent cells. The `Node` object has the `hash set` containers of its adjacent cells, nodes and edges. The C++ Standard Template Library(STL) is used for the `vector` and `hash set` containers.

In the present code, all pointers are implemented using the Boost shared pointer.[11] Since the shared pointer is automatically deleted if it

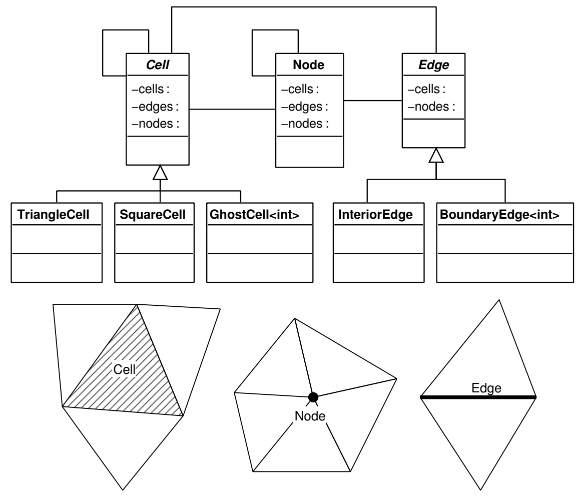


Figure 4: Class hierarchy and relations of mesh components.

is not refered from anywhere, the code is free from managing memory leak. Since the nubmer of compnents dynamically changes in AMR, the code becomes simple by the shared pointer. (Notice: since the mesh component circularly refers each other, the component must clear the relations at the coarsening process in AMR.)

Strategy Pattern[4]

Figure 6 shows the data structure of physical values. Primitive values such as density, pressure, velocity and etc. are handled through an abstract `Gas` class. Concrete classes such as an `IdealGas` class, `Air` class and etc. are derived from the `Gas` class. Each concrete class has its primitive values as data members. Operations particular to each gas species, for example, the calculation of speed of sound, are sepcilaized in each concrete class.

Since the present code is the cell-centered scheme, the `Cell` class has the shared pointer of the `Gas` class , `gas`, which indicates a concrete gas object. When a client class requests an operation about primitive values to the `Cell` class, the `Cell` class delegates the request to the concrete gas object through the `gas` pointer.

Since a client class need not know what class the `gas` points, the exchange of gas spcecies becomes very easy.

Application to a Mesh construction

Builder Pattern[4]

Mesh is constructed by the Builder Pattern. Figure 7 shows its class relations. The Builder Pattern is useful to create the object which is composed of different objects. Here, it is explained how to construct the mesh from an initial mesh data.

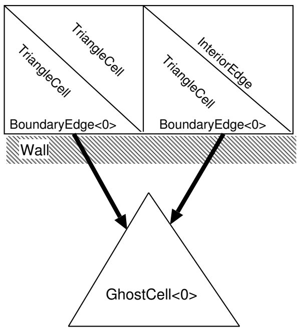


Figure 5: `GhostCell`.

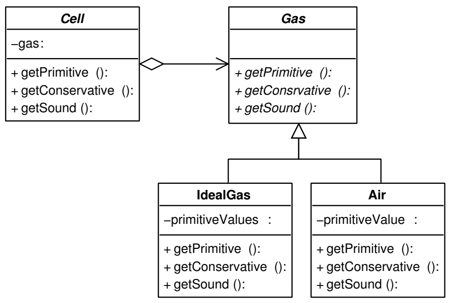


Figure 6: `Gas` class (Strategy Pattern).

In an initial mesh construction, a `meshConstruct` operation of a `MeshDirector` class is called. In the operation, a data file is set to an abstract `MeshBuilder` class, and then operations of the `MeshBuilder` class are called to construct mesh.

For example, a `buildCell` operation of an `InitMeshBuilder` makes new mesh components reading the mesh data file, and adds them to a `cells` data structure of a `Mesh` object.

By this pattern, the details of mesh construction are hidden in the concrete classes of the `MeshBuilder`. For example, when the method of mesh construction in a restart computation is different in an initial computation, the `MeshDirector` need only use the `RestartMeshBuilder` instead of the `InitMeshBuilder` class.

Factory Method Pattern[4]

In the `build` operation of the concrete classes of `MeshBuilder`, the Factory Method Patter is used to create a new object of the mesh component. Figure 8 shows its class relations.

For example, in the `Edge` creation, Concrete creator classes corresponding to concrete `Edge` classes are derived from a `EdgeCreator` class. Each `create` operation of the creator class returns a new correspondig `Edge` object. Each creator classes are registered to the data member, `creators`, of

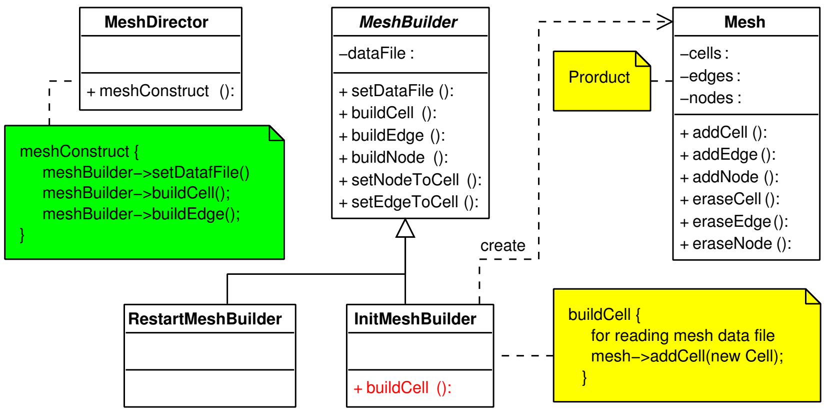


Figure 7: Mesh construction (Builder Pattern).

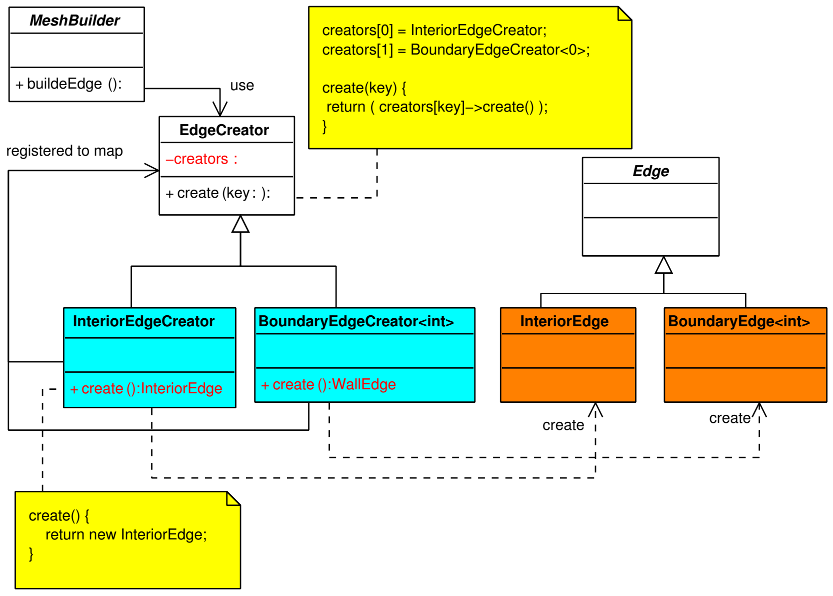


Figure 8: Factory Method Pattern.

a `EdgeCreator` class with its corresponding key. Then, a `create` operation of the `EdgeCreator` becomes to return the new object corresponding a key.

The `buildEdge` operation creates a new `Edge` object using the `EdgeCreator` by reading keys from a mesh data file. Since the `new` operations of C++ and `if-else` statements can be excluded from the code of the `buildEdge` operation, the code becomes simple and flexible.

The Factory Method Pattern can be easily implemented by Loki template library.[12]

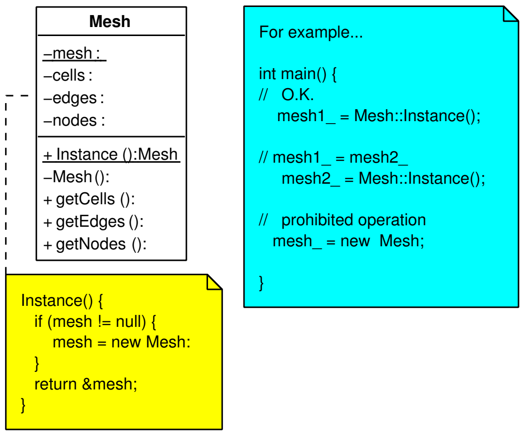
Hash set data structure

The insertion and deletion of mesh components frequently occur in the AMR. Then, a hash set data structure is used for the `cells`, `edges` and `nodes` data member of the `Mesh` class. The shared pointers of mesh components are registered to these hash sets by the `add` operations of the `Mesh` class. The hash set data structure can be easily used by STLport library.[10]

Singleton Pattern[4]

The CFD code must have only a `Mesh` object. In addition, the shared pointers of mesh components are refered from many objects through the `get` operations of the `Mesh` class. Hence, the `Mesh` object must be single and be globaly accessible.

A singlton `Mesh` class is shown in Fig.9. In the singlton class, its constructor is defined as a private member, and it has itself, `mesh`, as a static data member. Then, a exterior object can not create a `Mesh` object. When a static `Instance` operation is first called, a `Mesh` object is created. After second call, its operation returns the reference



```
Instance() {
  if (mesh != null) {
     mesh = new Mesh:
  }
  return &mesh;
}
```

```
For example...

int main() {
//   O.K.
    mesh1_ = Mesh::Instance();

// mesh1_ = mesh2_
    mesh2_ = Mesh::Instance();

//   prohibited operation
   mesh_ = new  Mesh;

}
```

Figure 9: Singleton Pattern.

of the `mesh`. By this mechanism, the singularity is ensured, and the mesh becomes globaly accessible by `Mesh::Instance()` because it is a static operation.

The Singleton pattern is used to design most classes except for mesh components, for example, classes about mesh construction, flux calculations, AMR and so on.

The Singleton pattern can be easily implemented by Loki template library.[12]

Application to an Inviscid Flux Calculation

Figure 10 shows the class hierarchy and relations of an inviscid flux calculation. A `visit` operation of a `FluxCalculator` calculates the flux on each edge, and adds it to the adjacent cells of the edge. In the `vist` operation, the inviscid flux is calculated using a `getPrimitive` operation of an abstract `Interpolation` and a `getFlux` opeartion of an abstract `FluxScheme`, as following,

```
visit(edge) {
  left=interpolation->getPrimitive(edge,0);
  right=interpolation->getPrimitive(edge,1);
  flux = flusScheme->getFlux(left,right)
  cells = edge->getCells();
  cells[0]->renewRHS(flux, 0)
  cells[1]->renewRHS(flux, 1)
}
```

The concrete classes of the `Interpolation` class calculate the left(0) and right(1) primitive value on the edge. The concrete classes of `FluxScheme` calculates the inviscid flux using these primitive value. Then, the flux is added(0) to or substracted(1) from the RHS member data of `Cell` objects by the `renewRHS` operation of the `Cell` class.

Since both `Interpolation` and `FluxScheme` class are designed as the Strategy pattern, the shcemes are easily exchangable.

In a concrete `MUSCL` class, a `getPrimitive` operation is implemented by the `getGradient` operation of a `GradientCalcualtor` and the `getLimiter` operation of a `LimiterCalculatior` class.

Besides, the `Vector` container of TVMET[13] library is used for the flux, primitive value and so on. The `Vector` container has the operations of vector-vector addition, matrix-vector product and etc., and its run-time is quitely fast for the tiny vector whose elements is almost less than 20.

Visitor Pattern[4]

Figure 11 shows the class relations of the Visitor Pattern for the flux calculation. The flux calculation is invoked of the `fluxCalculate` operation of a `CFD` class, which becomes a core class in the CFD code. A loop is conducted for the container which consists of several concrete Edge objects. However, the flux on boundary edges must be calculated by different methods. Then, the `fluxCalculate` operation may be designed to call the `visit` operation corresponding to a pointed concrete `Edge` object in a `if-else` statement with type checking. This loses the flexbility and readablity of the code.

The Visitor pattern is used to avoid this. The concrete `Edge` classes are designed to have accept operations which transfers itself to the corresponding `visit` operations of `FluxCalculator` class.

Figure 12 shows the sequence diagram of this pattern.

1. The `fluxCalculate` operation is called and a loop on the `edges` starts.
2. The `accept` operation is called for a shared pointer of `Edge`, and the `fluxCalculator` object is transfered to the concrete `Edge` object.
3. In `accept` operation, the concrete `Edge` object calls its corresponding `visit` operation of the `FluxCalculator` class.
4. The flux is calculated in the `visit` operation.
5. repeat 1 $\sim$ 4.

By this pattern, the code becomes simple though its run-time may be slightly slow. This pattern can be also implemented by Loki template library.[12]

Application to an AMR

In the present code, the Rivira's bisection[8,9] algorithm is used for AMR. The `Cell` objects need store its parent cell and son cell as shown in Fig.13. In addition, the data for AMR level and etc. becomes necessary. These data is stored in the

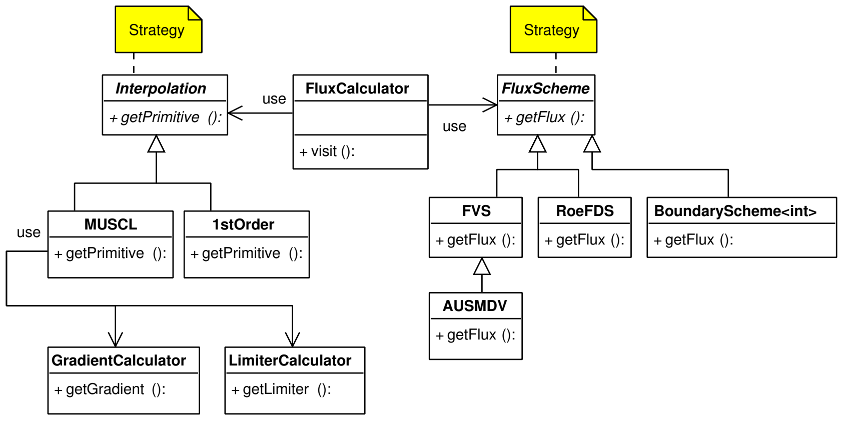
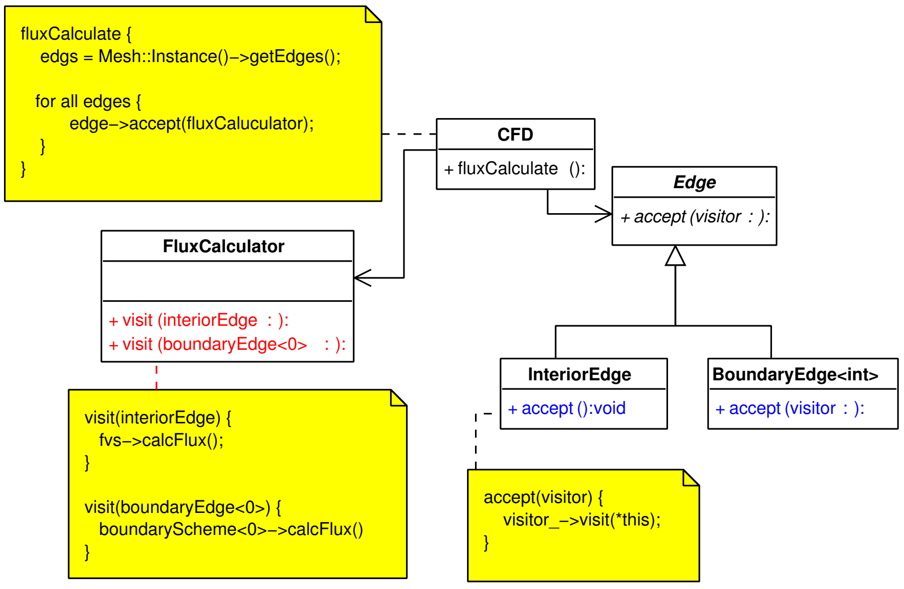


Figure 10: Class hierarchy and relations of Inviscid Flux Calculation.



```
fluxCalculate {
    edgs = Mesh::Instance()–>getEdges();

   for all edges {
        edge–>accept(fluxCaluculator);
   }
}
```

```
visit(interiorEdge) {
   fvs–>calcFlux();
}

visit(boundaryEdge<0>) {
   boundaryScheme<0>–>calcFlux()
}
```

```
accept(visitor) {
    visitor_–>visit(*this);
}
```

Figure 11: Class relations of Visitor pattern.

`AMRstate` class, sperated from the `Cell` class, to simplify the structure of the `Cell` class. Figure 14 shows the class relations of AMR. Classes conducting AMR refer and operate the `AMRstate` object through the `getAMRstate` operation of the `Cell` class. : The operation returns the shared pointer of the `AMRstate` object.

The Builder pattern is basically used for AMR. The `AMR` class directs the AMR through Builder classes, an `AMRMarker`, a `Refiner` and a `Coarsener`. The Builder classes renew the `Mesh` object by refering and operating the `AMRstate` objects and the `Mesh` object.

The `AMRMarker` object calculates an error indicator[8] and add refine- or coarsen-marked cells to the `refineCells` or `coarsenCells` of the `AMR` object.

The `Refiner` and `Coarsener` object executes the AMR and changes the `AMRstate` of each `Cell` by reading the `refineCells` and `coarsenCells`.

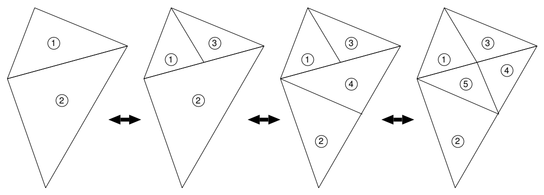


Figure 13: Bisection AMR.

The complex operations of the AMR are enclosed in each Builder class as its private operations which are used in each `do` operation.

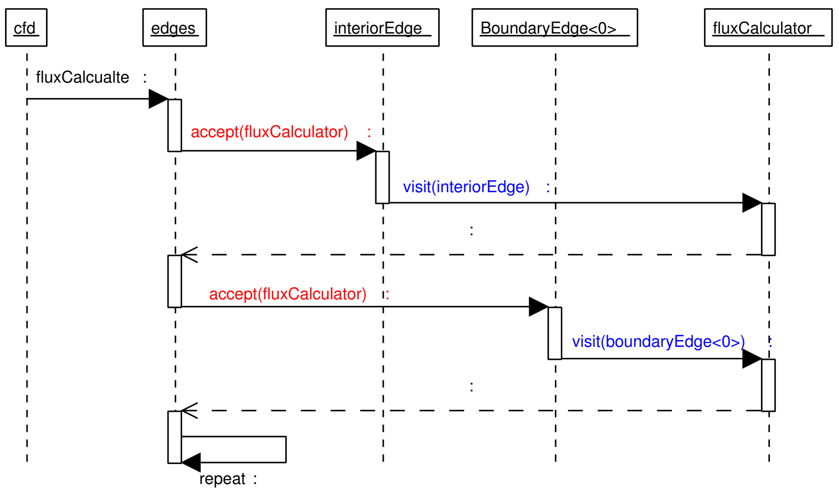


Figure 12: Sequence diagram of Visitor pattern.

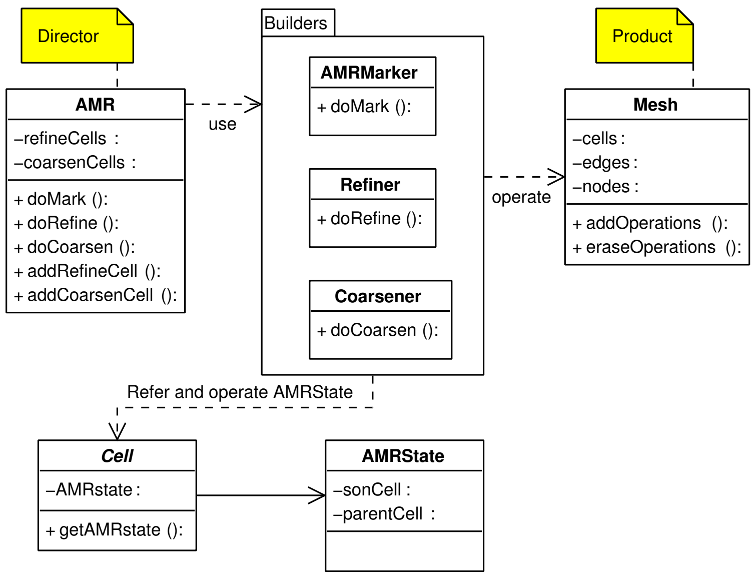


Figure 14: AMR class relations.

NUMERICAL RESULTS

Sod's Shock Tuble Problem

The Sod's shock tuble problem[14] is solved by the present AMR code. In this problem, trianglar cells are used. Figure 15 shows the density contours and mesh. The lines are exact solutions of the shock tube problem, and the circles are calculated densities. The calculated strength and speed of shock wave agree with exact solutions.

Comparison of run-time speed between Fortran and C++ code

To compare the run-time speeds of a Fortran and an OOP (C++) code, the Sod's shock tube problem is again solved using the square mesh (100×10 cells). In this computation, the AMR is not conducted. Table 1 shows the run-time speed of 1000 iterations. The computation is run by the Intel Fortran or C++ compiler 7.1 on a Pentium4 2.26GHz mechine. The compiler options are `-O3, -tpp7 -xW`.

Table 1: Run-time speed of 1000 iterations

	Fortran	C++
sec	1.83	12.96

The run-time speed of the C++ code is about 7 times as slow as the Fortran code. However, the flexibilty and reusability of the code is dramatically enhanced. For example, in the present test, the change of the code is only the exchange

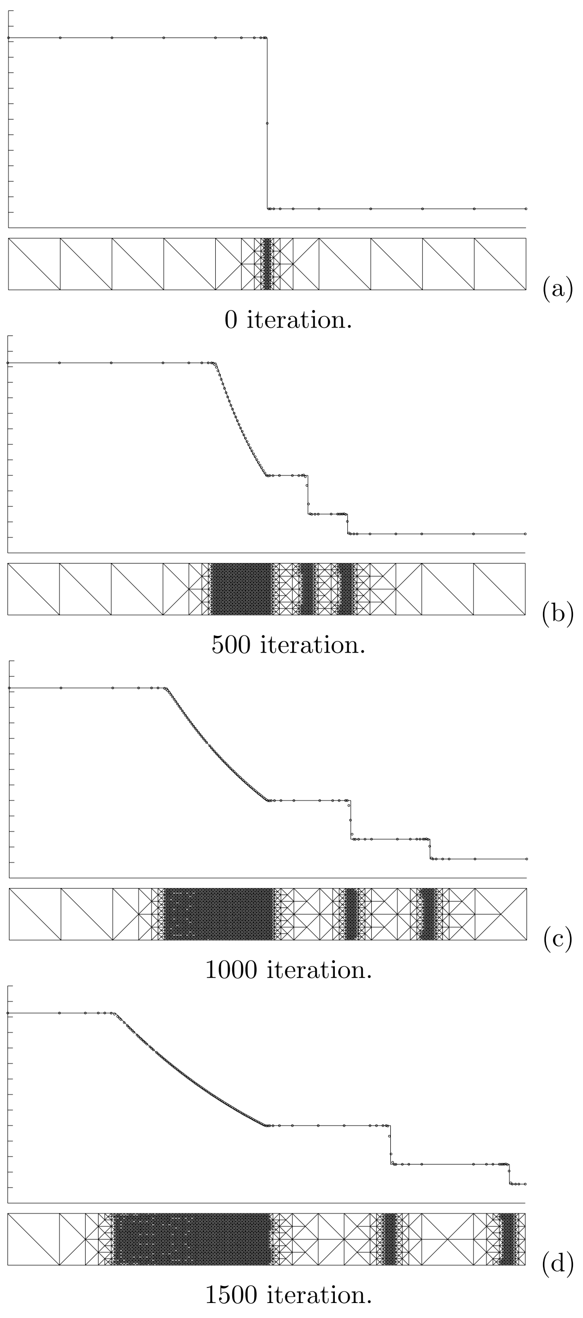


Figure 15: Adaptive mesh and denstiy contours in the Sod's shock tube problem.

of `Cell` type from the `TrangularCell` class to the `SquareCell` class.

SUMMARY

Object-Oriented design patterns to a general CFD code were suggested, and were also applied to an Adaptive Mesh Refinement(AMR) using unstructured mesh. For a test problem, the shock tuble problem is solved. In addtion, a run-time speeds of a Fortran and C++ code are compared. The run-time speed of the C++ code is about 7 times as slow as the Fortran code. However, the flexibilty and reusability of the code is dramatically enhanced.

ACKNOWLEGEMENT

The calss figures in this paper were drawn by "Poseidon for UML Commuity Edition," Gentleware AG.^{15}

REFERENCES

[1] Cambier, L. and Gazaix, M., "*elsA*: An Efficient Object-Oriented Solution to CFD Complexity," AIAA Paper 2002-0108, 2002.

[2] Tchon, K-F., "Object-Oriented Programming for a Temporal Adaptive Navier-Stokes Solver," AIAA Paper 95-0574, 1995.

[3] Stroustrup, B., "The C++ Programming Language, 3rd ed." Addison-Wesley, 1997.

[4] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, 1995.

[5] Unified Model Language^{TM}, "http://www.uml.org"

[6] Wada, Y. and Liou, M.S., "A Flux Splitting Scheme with High-Resolution and Robustness for Discontinuities," NASA TM-106452, 1994.

[7] Venkatakrishnan, V., "On the Accuracy of Limiters and Convergence to Steady State Solutions," AIAA Paper 93-0880, 1993.

[8] Sharov, D. and Fujii, K., "Three-Dimensional Adaptive Bisection of Unstructured Grids for Transient Compressible Flow Computations," AIAA Paper 95-1708-CP, 1995.

[9] Miyaji, L. and Fujii, K., "Simulation of unsteady shock wave reflections using adaptive unstructured grids," Proceedings of 15th International Conference on Numerical Methods in Fluid Dynamics, pp.334-339, 1996.

[10] STLport web site, http://www.stlport.org

[11] Boost web site, http://www.boost.org

[12] Alexandrescu, A. "Mordern C++ Design," Addison Wesly, 2001, http://www.moderncppdesign.com

[13] Petzold, O. "Tiny Vector Matrix library using Expression Templates," http://tvmet.sourceforge.net

[14] Sod, G.A. " A survey of several finite differtization of parabolic differential equations in one space variable, Journal of Computational Physics, vol.43, pp.1-31, 1978.

[15] Gentleware AG, http://www.gentleware.com