Re-engineering Legacy Code with Design Patterns: A Case Study in Mesh Generation Software

Chaman Singh Verma
Dept. of Computer Science
The College of William & Mary
Williamsburg, VA 23185
csv@cs.wm.edu

Ling Liu
Dept. of Computer Science
The College of William & Mary
Williamsburg, VA 23185
lingliu@cs.wm.edu

***Abstract*— Software for scientific computing, like other software, evolves over time and becomes increasingly hard to maintain. In addition, much scientific software is experimental in nature, requiring a high degree of flexibility so that new algorithms can be developed and implemented quickly. Design patterns have been proposed as one method for increasing the flexibility and extensibility of software in general. However, to date, there has been little research to determine if design patterns can be applied effectively for scientific software. In this paper, we present a case study in the application of design patterns for the re-engineering of software for mesh generation. We applied twelve well-known design patterns from the literature, and evaluated these design patterns according to several criteria, including: flexibility, extensibility, maintainability, and simplicity. We found that design patterns can be applied to significantly improve the design of the software, without adversely affecting the performance. As a secondary practical contribution, we believe that this research can also lead to the eventual development of a flexible framework for mesh generation.**

Keywords: Design patterns, generic programming, mesh generation.

I. Introduction

Software reuse is identified as one of the best strategies to handle complexities associated with development and maintenance of complex software. Reuse has been very successful in many areas, especially in compilers, operating systems, numerical and GUI libraries for a long time. Although many libraries have passed the test of time, they suffer from one big disadvantage: they have fixed interfaces and data structures. There is very tight coupling between their algorithms and data; therefore these libraries are not extendable for user defined data types.

Today, design patterns [1] and generic programming [2] are emerging techniques which have been proposed as solutions which can alleviate this problem. Design patterns stress upon decoupling the system for increasing flexibility, and generic programming allows developers to reuse the software by parameterizing the data types. Many application domains (e.g. GUI builders, network communication libraries) have greatly benefited from using design patterns.

Some research has been performed in the use of these methods for the development of industrial software. For example, Coplien [3] *et. al.* have provided industrial experience with design patterns.

However, while generic programming is a well-established practice in scientific software, today we lack evidence that design patterns can significantly improve such code without adversely affecting performance.

In this paper, we explore how design patterns can be applied to re-engineer legacy code to increase the flexibility of the system. We have applied twelve design patterns from the literature [1] to an existing mesh generation software system. We characterized the design patterns in terms of three primary design criteria: static and dynamic extendibility, reliability, and clean design of the system.

We then evaluated the resulting system in terms of these criteria. Our evaluation assumed that users of software implemented in an object-oriented language are willing to sacrifice some performance for other benefits. As a result, our evaluation of the performance impact of the design patterns is informal, and is meant to ensure that any performance degradation is acceptable to users.

We characterize the use of each design pattern in our system in terms (1) the probability of being able to apply it, (2) the benefits of using it, and (3) the extent to which the code must be changed to implement it. We conclude, based on our experiences, that the modified system exhibits enhanced flexibility, extensibility, maintainability and understandability, without sacrificing too much performance.

As a longer term goal, we hope to use design patterns to develop a flexible framework for mesh generation. This framework will allow researchers to collaborate on the development of new algorithms and data structures for mesh generation, and to perform experiments to assess the quality of existing algorithms. In addition, we believe that this framework will lead to the development of a web-based service-oriented version of the software.

The rest of the paper is structured as follows. In Section II we give background and related work, Section III describes re-engineering legacy code with design patterns, in Section IV we evaluate our work and Section V concludes.

II. Background

Parnas [4] explained some realities about software aging. Developing reliable and robust software is a difficult and time consuming human activity. Most legacy software evolves over a large period of time. Such software is trustworthy, in its limited functionality. Today, much software is still being used because the user base is very large, and because the software contains hidden and critical design decisions. According to Parnas, software aging is inevitable, but efforts must to taken to delay the degradation. Instead of throwing away such software, we need some solution which allows us to use it in our new system, and then to slowly change or replace it as our system evolves.

Re-engineering, as defined by Chikofsky and Cross [5], is the examination of the existing legacy software in order to understand its specification, followed by subsequent modification or re-implementation to create a new, improved form. To date, a lot of research has been done in providing tool support for software re-engineering. For example, Verhoef in [6] discussed the necessity for automating modifications to legacy assets. Brunekreef [7] also presented a software renovation factory which is user-controlled through a graphical user interface.

In this work, we attempt to manually re-engineer a legacy system into a more extendable and adaptable system. By using design patterns, we hope to be able to use legacy code in new software, and also re-engineer it in order to improve characteristics such as the flexibility of the resulting system.

In this section, we first present some of the disadvantages of legacy software, and then analyze the causes of inflexibility. Next we propose some explanations for the lack of use of reusable software. Finally, we discuss the requirements for making software adaptive.

A. Disadvantages of Legacy Code

There are several disadvantages of using legacy code.

- It is difficult to maintain and extend the functionality of most legacy software, especially if the software is written in functional languages such as C and FORTRAN.
- Rewriting them requires a large investment of money and human effort.
- Such software contains substantial duplication of code for the same functionality, where the code differs only in the data types.
- Legacy code does not take advantage of modern processor design. Most of the code was written when thread programming was in its infancy and distributed computing was non-existent.
- In general, most legacy code handles memory and errors poorly. For example, FORTRAN does not have dynamic memory allocation and C code often has memory leak problems.

B. Analysis of Legacy Code Inflexibility

Before we begin to re-engineer legacy code, we need to understand the primary causes for its inflexibility.

- *Conditional statements:* "If-then-else" and "switch" statements are fundamental to almost all programming languages, but their use sometimes restricts extension because hard-coded constructs simply assume that the alternative conditions are *finite* and remain *fixed* throughout the lifetime of the software. If the conditions or requirements change, adding new conditions require significant effort. Object-Oriented programs always try to eliminate the use of switch statements.
- *Multiple inheritance:* Although C++ allows multiple inheritance, in general it creates more complexities and ambiguities than the solutions it provides. The problem with multiple inheritance is the famous *Diamond Problem* [8]. Some programming languages such as JAVA have already discarded this feature in favor of simplicity and consistency, using single implementation inheritance and multiple interface inheritance.
- *Lack of abstraction:* Object-orientation is a powerful technique as long as we are able to break down the systems into smaller granularity and appropriate objects. There are no silver bullets in using inheritance and polymorphic features of object-oriented programming—even though the features are present, it is difficult to use them to implement the proper abstractions for a given system. As a result it is not uncommon to find much duplication of concepts and functions in a given system.
- *Lack of separation of concerns:* Software has three basic components, namely: concept (what you want to do), algorithm (how to do it) and data management (how to manage data and resource). Parnas [4] demonstrated the importance of modularity, and gave criteria for decomposing a system into modules of autonomic concerns. Unfortunately, even after 30 years since the publication of this seminal paper, most applications developed today still have tight coupling among concerns, so it is difficult to change or replace any part of the code. The Standard Template Library (STL) is the first widely used software library which separates these three concerns. For example, STL provides abstractions for containers, iterators for containers, and algorithms over containers. Each of these is largely independent of each other.
- *Conservative assumptions:* Most programmers implement the code considering only the immediate requirements, and few believe that their programs will have a very long lifetime. As a result, they make certain assumptions in their implementations which become obsolete very quickly.

C. Reluctance for Reusing Software Components

Despite the enormous advantages of reusable software components in both the short and long term, incorporating them into new systems or in restructuring the existing applications have not been up to expectations [9]. Reluctance could be attributed to some of the following reasons:

- It is hard to manually understand the behavior of the code or side-effects which may be introduced as a result of using the software. In addition, automatic or semi-automatic tools for analyzing these effects are inadequate.
- An incremental approach to software reuse is also difficult and error-prone. Sometimes, small changes are just

not possible. As a result, either we do not change the software, or we change the entire system.
- There is much uncertainty on the part of software developers as to whether reuse will significantly improve the quality of the resulting system. There is little evidence and few accepted metrics for success in the reuse of software in real applications. Very often, there is often a trade off between performance and quality.
- The learning curve could be steep.
- Highly motivated software developers are tempted to rewrite code.
- Old systems often have little or no documentation.
- If the reusable component comes from a commercial company, there might be issues related to patents, copyrights and royalty payments.

For a comprehensive introduction to software components and reuse, see [10].

D. Adaptive Software

We hope to develop new software or re-engineer legacy codes into software which is adaptable. Adaptive software has the following characteristics:

- *Program for change:* Although it is hard to predict the future, objects should not make assumptions which are valid for only a short duration of time. Whenever possible, a good design should abstract some core concepts into a small number of functions and classes, and provide simple interfaces to access the functionality.
- *Flexible and dynamic relationships:* Rarely an object exists in isolation. There has to be a simple mechanism to create permanent and temporary relationships among objects.
- *Centralized authority:* Programs are difficult to understand, maintain and extend when some decision or functionality is scattered throughout the code. Whenever possible, there should be one place for one piece of functionality. This simplifies modification and testing of the system.
- *Division of labor:* A class should have a single, well-defined purpose as well as a simple interface. A class should delegate other responsibilities to other suitable classes. Minimization of functionality increases both the productivity, reliability and reuse.
- *Standardization:* Successful software reuse requires standardization. With standardization comes reliability, easy availability and large support.

III. Re-engineering Legacy Code

According to Gamma *et. al.* [1] design patterns are recurring solutions to software design problems which we find repeatedly in real-world application development. When we use design patterns, we do not reinvent the wheel. Another way of looking at design patterns is to consider them as well-proven component integrations with a common vocabulary for the system designer and developer. Buschmann [11] collected design patterns in the context of software architecture.

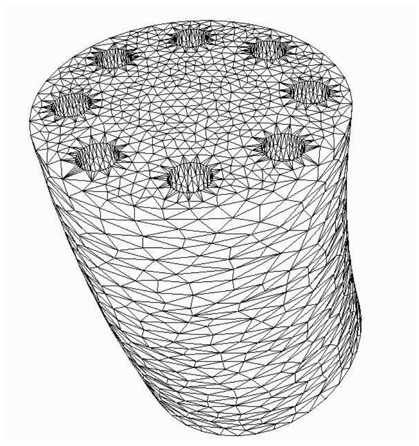


Fig. 1. Surface mesh generation on pipe

Gamma *et. al.* [1] identified 23 design patterns and created a catalog, which is known as the GoF *(Gang of Four)* book. We have taken several patterns from this catalog and applied them to our application. In the figure 2 we list all the GoF design patterns which we applied, and the main purpose behind using each pattern in our application. To shorten our paper, we have not shown any examples of some patterns in the next section(Singleton, Reference counting, Decorator, Facade etc). Although design patterns are often written in an object-oriented language, design patterns have little to do with object-orientation ([3]).

Application: Mesh Generation

Numerical simulation uses partial differential equations (PDEs). For example, the Navier-Stokes equations are used in Computational Fluid Dynamics. The first step in numerical simulation is to discretize the geometric space into a large number of cells. In 2D these cells are triangles or quadrilaterals, and in 3D they are tetrahedra, pentahedra or hexahedra. Once a good quality mesh has been created, numerical discretization of the PDEs is carried out, and for each cell, governing equations are solved. For complex geometries, an unstructured mesh (in which the topology is explicit) is preferred because of the engineering requirements for high quality mesh. A sample mesh generated over a simple geometry is given in Figure 1.

A. Components in Mesh Generation Software System

Mesh generation is a fairly complicated process which utilizes many external libraries, software tools, algorithms and data management tools. The following are main components in mesh generation which explains the need for reusing the software:

- *Geometric modeling:* Construction of a geometric model involves designing the model with geometric primitives such as circles, lines, planes or NURBS (Non-Uniform Rational B-Splines) curves and surfaces. Highly interactive graphical display systems are needed to design complicated models, which is often done with commercial CAD systems.

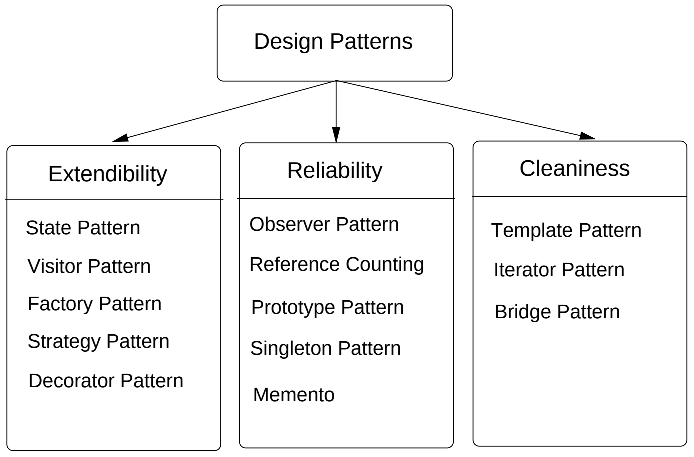


Fig. 2. Design patterns objectives

- *Adaptive or Multi-precision library:* Geometric algorithms demand robustness in numerical calculation. Most of the time, standard IEEE floating points are not suitable for this task, and therefore researchers either use libraries for exact arithmetic or fast adaptive multi-precision computation.
- *Geometric kernel library:* A geometric library is a collection of large spatial data structures for geometric space (e.g. Kd-trees, quadtree, octree, BSP, etc.). These libraries often provide algorithms for computing the convex hulls, 2-3D triangulations, Voronoi diagrams and fast proximity queries.
- *Mesh generation algorithm:* These libraries include components for generating a structured or unstructured mesh in the specified geometries. An unstructured mesh is mostly generated by using either *Advancing Front* or *Delaunay Triangulation* algorithms.
- *Sequential and parallel data structures:* Very often, we need an extremely refined mesh containing millions of cells for finite element analysis. In order to provide efficient insertion, removal or query for some elements, commercial software often uses a database (e.g. SQL or Oracle).
- *Domain decomposition and object migration tools:* We often use parallel processing to reduce the time and memory requirements for the execution of an application. Software components are needed to decompose, distribute and control the distributed tasks.
- *Interactive visualization:* Interactive graphical systems help in understanding and modifying the geometric space and mesh generation. In fact, they are an integral part of the mesh generation process.

B. Applying Design Patterns

In the following section, we apply several GoF design patterns in our application and explain why they are needed using small examples.

1) **Adapter Pattern** Sometimes there are incompatible interfaces between two software components. Adapter pattern provides a clean mechanism to adapt one interface to another. Adapter pattern could also be used to hide the old design with the new one without reimplementing the class from scratch. The end user will perceive the class according to new design rules.
In our application, the geometric modeler uses NURBS curves and surface which were originally written in ANSI C. Here is how we wrap the original code in the new class

```
namespace NURBS {
class Curve
{
pubic:
   Curve( NURBS_Curve_t *c );

   Point2D  evaluate(double t );
      point_t pt = NURBS_EvalCurve( oldcurve, t);
      Point2D   result;
       result[0] = pt.x;
       result[1] = pt.y;
    }
 private:
    NURBS_Curve_t *oldcurve;
   };
   }
```

In this example *NURBS_Curve_t* class is an old structure which is not consistent with the new system. Old structure and old functions are kept as private member of the class. The end user can use the new system which uses the old system without ever knowing inner details of the old system.

2) **Bridge Pattern** Information hiding is fundamental to OOP. Keeping class abstraction from its implementation has many advantages for the following reasons.
   - Most of end users are only interested in using the classes, and not their implementations.
   - Keeping implementation in header files results in longer compilation time and if the header file changes, the entire application has to be recompiled.
   - It makes changing implementation easy. (There may be different implementation for different platforms)
   - Many classes can use reference counting for lazy object copying.

The Bridge Pattern provides the solution to the problem by providing a pointer to the representative class in the original class and forwarding all the requests from the main class. The only disadvantage of this approach is that it require indirection for every function call, but this is the price we are willing to pay for increasingly the flexibility.

```
//Implemented in filename MeshGen2D.h
class MeshGen2D
{
public:
  MeshGen2D() { rep = new MeshGen2DImpl(); }

  void setData( int d)
```

```
       { rep->setData(d); }
  int  getData() const {
       { return rep->getData(); }
private:
   MeshGen2DImpl *rep;
};

// Implemented in filename MeshGen2DImpl.h
class MeshGen2DImp
{
public:
   MeshGen2DImpl();

   void setData( int d) { data = d; }
   int  getData() const  { return data; }

private:
  int  data;
};
```

This pattern is used in the new system wherever a class implementation is lengthy and changable.

3) **Factory Pattern** Consider the following code from legacy code

```
void  Reader:: readFile( ifstream &infile)
{
     GeoEntity *geoEntity;
     while(infile) {
          infile >> objectType;
          switch( objectType )
          {
            case 0:
                 geoEntity = new GeoVertex;
                 break;
            case 1:
                 geoEntity = new GeoEdge;
                 break;
            case 2:
                 geoEntity = new GeoFace;
                 break;
            case 3:
                 geoEntity = new GeoCell;
                 break;
           }
      }
```

Although the code seems to be clean design, there are some shortcomings with this style of object creation which becomes problematic in future. Consider the following situations

- We may want to use some customized allocators for performance improvement.
- We may want to add error reporting messages if the allocation fails.
- We may want to hook some functions whenever we create new instance of an object.
- We may want to add new shapes.

Factory pattern provide a solution for this problem.

```
  Factory<GeoEntity>  factory;

  factory.Register( 0, GeoVertex::create);
  factory.Register( 1, GeoEdge::create);
  factory.Register( 2, GeoFace::create);
  factory.Register( 3, GeoCell::create);

   GeoEntity *geoEntity;
   while(infile) {
        infile >> objectType;
        geoEntity = factory.newProduct( objectType );
   }
```

Where *create* is a static member function for creating a new object.

With this pattern, user is relieved forever from hard-coding new object creation. He can register or unregister product using the services provided by factory. Another advantage of using *create* member function for every object is that this function can be modified, extended for various purposes without changing the application.

4) **Memento Pattern** There are many situations where we want to store the internal representation of an object, for example:

- **Transmitting objects over network:** To transfer objects across the network, sender has to pack the data into single contiguous buffer (marshaling) and reconstruct the object at the receiver side. (unmarshalling).
- **Persistent Storage:** We want to store the object into persistent storage for future use.

For ordinary structures and simple data types, serialization is simple. Memento pattern is very useful when

- we do not have access to the private data of a class.
- the classes we use are in the library form and we do not have access to source code.
- we want to override default (un)marshalling functions to store only part of the information instead of entire class data.

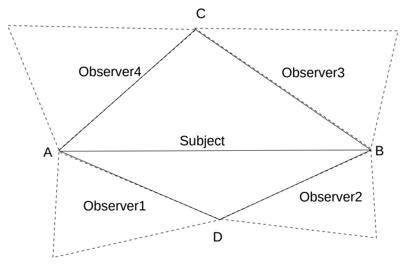
```
1 hash_map<int, Face*>   facedb;
2 Memento<hash_map<int,Face*>  > memento(facedb);

3 vector<char> buf = memento.setState();
3 MPI_Send(&buf[0], buf.size(), MPI_CHAR, dest,
         0, MPI_COMM_WORLD);
4
5
6 MPI_Recv(&buf[0], numrecv, MPI_CHAR, source,
7        0, status, MPI_COMM_WORLD);
8 facedb = memento.getState(buf);
```

In the line 2, we serialize the object and store in the memento object and in the line 8, de-serialization take place using memento class.

5) **Observer Pattern** Object rarely exists in isolation. Whenever state of an object changes, sometimes it is necessary to notify its dependent or peer objects, so that they can take appropriate actions. In figure 3 an edge AB has been flipped to CD therefore lots of changes take place Edge BD and AC have now new triangles as neighbor.

```
  // Adding observers of an edge AB.
  edgeAB->addObserver( edgeCB );
  edgeAB->addObserver( edgeAC );
  edgeAB->addObserver( edgeAD );
  edgeAB->addObserver( edgeDB );
```



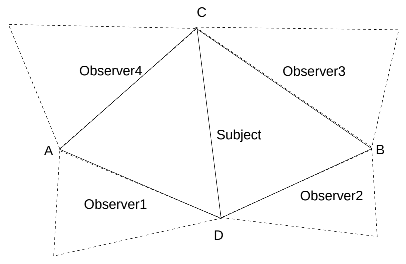


Fig. 3. Using Observer pattern in edge flip operation

```
.
// notify all the observers.
edgeAB->notify();
```

In the original code, complex data structure were used to reflect the changes whenever an edge is flipped. (We have omitted giving original code because of length considerations), but our experience says that this pattern was able to reduce coupling between the object that change and objects that needs change modifications. The resulting code is much cleaner and easy to understand. Observer pattern is a very powerful and useful pattern. When the subject changes, it notifies to the observers and they perform some calculations because of the changes. This pattern allows those calculations to be performed on *On demand* basis. To implement this change requires good understanding of the legacy code.

6) **Prototype Pattern** It is a pattern for object creation in which an object is responsible for creating a new object by cloning itself. Initially all the cloned objects inherits all the attributes from parent but these can be changed after the objects are created. Here is an example from our legacy code

```
Face* createNewObject( Face *face )
{
   Face *newface;
   switch( face->getType() )
   {
   case TRIANGLE:
        newface = new Triangle();
        break;
   case QUAD:
        newface = new Quadrilateral();
        break;
   case POLYGON:
        newface = new Polygon();
        break;
   }
   return newface;
}
```

The main difficulties with this approach are

- it is using switch statements to identify the type of a parent object which is hard to evolve.
- It has an assumption that only three kinds of object will be supported. Any new type of object will require adding one more *typeid* and changing this part of code.

Now if the prototype pattern were used we could use the same function as follow

```
Face* createNewObject( Face *face )
{
    return face->clone();
}
```

which is more precise and elegant. In order to use this pattern, every class has to provide a clone member function, which is simple and produced no side effects. In our re-engineered code, we used this pattern everywhere in the code.

7) **State Pattern** Most times, an object action depends on the type of input it receives. The most common approach is to use switch statement and invoke appropriate actions. Here is an example from the legacy code.

```
switch( cavityState)
{
  case 0:
       goodCavity();
       break;
  case 1:
       lockedCavity();
       break;
}
```

Depending upon the state, user take some actions. In many cases, number of states could be large and user may add or delete some states in the future releases. Using switches makes the code difficult to change. We apply state pattern to solve this problem in an elegant way. The following three steps are required

- Create state objects for each of the possible state derived from *State* abstract base class
- Assign unique integer ID to each state class and register them to State-Manager
- replace the switch statement by passing *state-ID to the state-Manager*

We create an object for each possible state, which is derived from state abstract class and register them into state repository as shown below.

```
class CavityState: public State
{
 public:
       void Operation();

protected:
   Grid   *grid;
   Cavity *cavity;
}
class GoodCavity : public CavityState
{
 public:
       void Operation();
}
```

```
class LockedCavity : public CavityState
{
  public:
       void Operation();
}
int main()
{
    CavityState   *cavityState;
    cavityState->Register(1, new GoodCavity);
    cavityState->Register(2, new LockedCavity);
    currentState = cavityState->getState(num);
    currentState->Operation(num);
}
```

8) **Strategy Pattern** Many time, we apply different algorithms for different input instances and conditions because some algorithms are well-suited to some specific input or requirement. If the number of algorithms are large or likely to change in future, it is not a good idea to hard-code them using switch statements. Here is an example from our legacy code

```
    switch(algorithm)
    {
    case 0:
         applyDelunayMethod();
         break;
    case 1:
         applyAdvancedFrontMethod();
         break;
    case 2:
         applyQuadtreeMethod();
         break;
    }
```

The flexibility of changing algorithm at run time and experimenting with different algorithms is important for the quality of the software output. The above method, although correct is not elegant. With strategy pattern we can add and choice different algorithms at run time.

```
class DelaunayMethod: public Strategy
{
 public:
     void applyAlgorithm();
}

class AdvancedFrontMethod: public Strategy
{
 public:
     void applyAlgorithm();
}

class QuadtreeMethod: public Strategy
{
 public:
     void applyAlgorithm();
}
int main()
{
    MeshGen2D  *meshGen;

    algRepository->Register(1, DelaunayMethod );
    algRepository->Register(2, AdvancedFrontMethod );
    algRepository->Register(3, QuadtreeMethod );

    currentStrategy = algRepository->getAlgorithm(1);
    meshGen->currentStrategy->applyAlgorithm();
}
```

This solution has the following advantages

- It has not used hard-coded switch statements.
- User registers algorithms in a repository and if needed, he can query the algorithm. This allows collaboration among team members and flexibility in choosing the algorithm appropriate to the requirement.
- The code is modularized into small number of classes which can be independently changed or tested.

9) **Template Pattern** Template pattern is so fundamental to object orientation that it is surprising to know that GoF classified it under patterns category. This pattern is also ambiguous because of the fact that C++ now support templates. (We wish that GoF could find a better alternate name to distinguish it from powerful C++ templates).

In the template pattern, some functions which are common to the subclasses are put into base class and a default behavior may be implemented. Derived classes can override this function and refine the behavior. The simplest example comes from base class object

```
class Object {
public:
  Object() {}
  virtual ~Object() {}
  virtual int hashCode() { return 0;}
  virtual Object* clone() { return NULL;}
  virtual bool equals(Object* obj)
               { return 1;}
  virtual const char* getName()
               { return "Object";}
}
```

Every class which is directly or indirectly derived from Object class can provide override function (such as hashCode, clone etc).

10) **Visitor Pattern** Consider the following part of the code

```
class Face
{
public:
  .
  void   getArea();
  void   getAspectRatio();
private:
  double area, asr;
}
```

Here *Face* is a abstract class for the different type of faces (triangles, quadrilateral ...) and with each face we have difference quality parameters. This is not a clean design. Suppose we change this class to

```
class Face
{
public:
  .
  void   getQuality();
private:
  double quality;
}
```

Where *quality* could be area or aspect ratio or any other user defined value associated with each face. With this implementation, quality is defined external to the class and can be defined by the user as

```
void FaceArea( vector<Face*> facedb)
{
 vector<Face*> iterator iter;
 Face *f;
 for(i = facedb.begin(); i != facedb.end(); ++i)
    switch( (*i)->getType() )
    {
     case TRIANGLE:
         Triangle *tri =
                  dynamic_case<Triangle*>(*i);
         tri->setQuality(TriangleArea(tri));
         break;
     case QUAD:
         Quadrilateral *quad =
               dynamic_case<Quadrilateral*>(*i);
         quad->setQuality( QuadeArea( quad ));
         break;
    }
  }
}
```

Well, this code will work, but all the elegancy of object-orientation and simplicity are hardly visible. Such codes are difficult to maintain.
We solve this problem using *Visitor Pattern*

```
class Grid
{
 public:
     void  accept( Visitor<Face> *v ) {
     for( int i = 0; i < facedb.size(); i++)
            v->visit( facedb[i] );
     }
private:
   vector<Face*>  facedb;
}

class AreaVisitor : public Visitor<Face>
{
 public:
   void visit( Face *f ){
        double q = getArea(f);
        f->setQuality(q);
   }
 private:
   double getArea( Face *f);
};
class AspectVisitor : public Visitor<Face>
{
 public:
   void visit( Face *f ){
       double q = getAspectRatio(f);
       f->setQuality(q);
   }
 private:
   double getAspectRatio( Face *f);
};

int main()
{
  Grid2D  g2d;

  Visitor<Face> *varea = new AreaVisitor;
  grid.accept(varea);

  Visitor<Face> *vasr  = new AspectVisitor;
  grid.accept(vasr);
}
```

With this pattern, we are able to redefine the functionality of the class without changing it. Since this functionality is outside the class, it is very easy to extend by creating a new visitor class.

11) **Iterator Pattern** There are large number of data structures (vector, tree, graph, link list etc) to store collection of objects. A particular data structures is decided by the applications in hand. Iterator pattern provides technique by which we can access elements of a container without exposing its internal representation.
Although GoF provides a simple Iterator pattern, in our view C++ iterators are more powerful, and we do not see any reason why they should not be directly used instead of GoF pattern. The following program tells how we do it.

```
Class Grid1D {
  typedef multimap<int,Edge*>  Container;
public:
  typedef Container::iterator edge_iterator;
  .
  .
  Edge*  currentItem(edge_iterator iter)
           {return iter->second;}
  .
private:
    Container  container;

}

int main()
{
    Grid1D  *g1d;
    Grid1D::edge_iterator   eiter, ebegin, eend;

    ebegin = g1d->edges_begin();
    eend   = g1d->edges_end();
    for( eiter = ebegin; eiter != eend; ++eiter) {
         Edge *edge = g1d->currentItem(eiter);
 .
 .
     }
```

The application does not need to know anything about container used in the class. In future, if we decide to change “multimap” container to “hash_map”, only one line in the header file will change which is a local change. There is no need to change anything in the user application.

C. Using Generic Libraries

Most of the legacy codes make use of data structures such as link-list, vector, hash table, etc in their code, With the availability of Standard Template library (STL) and related Boost C++ libraries, these data structures can easily be replaced by standard data structures provided by these libraries. Since

STL was designed keeping performance in mind, only very few software may need customized libraries of much higher performance.

```
void  DoSomething( Grid1D* g )
{
   double *buf = new double[g->numNodes()];
   .
   .
   delate buf;
}
```

instead of using conventional arrays, if we use STL vector, we can avoid using delete every time (and avoid accidental memory leaks);

```
void  DoSomething( Grid1D* g )
{
   vector<double> buf;
   buf.resize(g->numNodes();
   .
   .
}
```

Other than standard data structures provided by STL, Matrix Template library(MTL), Iterative template library (ITL) and Boost Graph library(BGL) are some of the non-standard but very flexible and powerful libraries based on the STL design principle. The use of these libraries not only increases the reliability but also decreases the size of original code considerably. In our future studies, we plan to include them and undertake performance studies.

Arrays and char string are perhaps the most common in legacy code which are second class object. Using their first class equivalents such as vector and string in C++ STL, provides flexibility and reduces redundancy in the original codes.

IV. Evaluation of Efficacy of Design Patterns

Applying design pattern is tricky and sometimes difficult. We can justify effort only when we see some quality improvement in the new system. In this section, we answer some of the common questions.

- *Is new system more flexible ?*
  Yes. With the Prototype, Factory and abstract Factory, instantiating new objects has become very simple and flexible. With Strategy pattern, adding/replacing new algorithms has become very simple.
- *Is new system better maintainable ?*
  We follow the software maintainability defined by Fenton [12] Maintainability = Understandability + Modifiability + Extendibility + Testability
  With the above definition, design patterns are good for software maintenance. They enforce modularization which are easy to test than monolithic classes, we can modify a component without having side effects, extendibility is the prime motivation of patterns.
- *Is the process incremental ?*
  In general, no. Some of the design patterns such as factory and prototype pattern are very simple to implement. State and strategy patterns are relatively harder and require good understanding of the software. Observer pattern's full potential can be realized only when we understand nuts and bolts of the software. We are not sure whether reference counting could be carried out incrementally. Figure 4 we have listed probability of finding patterns in a typical scientific software. The most powerful patterns are at the bottom of pyramid, therefore most of the codes will be suitable for re-engineering with design patterns.
- *Are GoF pattern suitable for scientific computing ?*
  Yes, most of our software are experimental in nature and therefore have high degree of changeability. With design patterns we are able to add new features, and experiment with new algorithms.
  While we did not perform quantitative analysis of the performance impact of our changes, we did informally check that the performance did not degrade substantially. We did this by re-generating the mesh in Figure 1, which took at most 5 percent longer than the legacy version.
- *Is design pattern a good lingua franca ?*
  Yes. With design pattern we can explain the behavior, concepts and architecture of the software to both team member and to the seniors.
- *Are GoF patterns concise ?*
  Largely yes, but it seems that Memento, Template and Iterator patterns are just syntactic sugar patterns. Most of the developers use them without knowing that they are patterns.
- *Do we need new patterns to increase the flexibility ?*
  Yes. Similar to State, Factory and Strategy patterns, one of the big obstacles in reusing the software comes from using various termination condition. Consider the iterative solvers in linear algebra, there are many criterion to stop the iteration process and most of the software use predefined conditions. A *Conditional Pattern* may be a good choice. We also find that there are no good patterns for error handling and testing software. Since these essential parts of any software development, we need to find good patterns to address these recurring problems.

Overall, design patterns have significantly improved the quality of the software. They have forced modularization (State, Strategy, Visitor). The bridge pattern allowed us to keep implementation separate from interface. The Iterator pattern provided a consistent and simpler interface for traversing over the container. Design patterns helped us to evolve the legacy software toward a reusable, object-oriented design.

A. Design Pattern Mining

In large complex legacy code, finding the design pattern requires good understanding of the code. Ferenc [13] *et. al.* has reported developing automatic tools for finding patterns from UML graphs, but as of now we did not find any freely available tool on linux or other Unix platforms. For the time being, we explored them manually and noticed that in non-numerical scientific application, there exists possibility of ap-

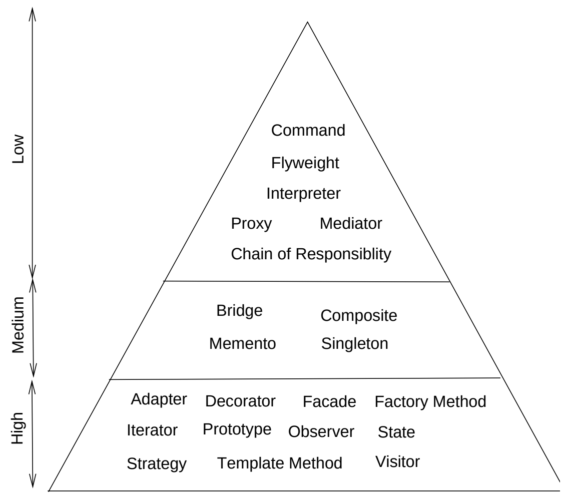


Fig. 4. Probability of finding GoF design patterns in legacy codes

plying design patterns to improve the quality. Most of scientific applications employ different algorithms for different input, use different data types and have dependency among objects. During re-engineering process, the use of design patterns involves decision about level of intrusion in the software. Table IV-A provides a guideline about level of intrusion which could help in taking decisions.

Pattern	Low Changes	medium Changes	Larges Changes
Adapter	-	√	-
Bridge	√	-	-
Factory	√	-	-
Memento	√	-	-
Observer	-	-	√
Prototype	√	-	-
Singleton	√	-	-
Strategy	-	-	√
State	-	-	√
Template	-	√	-
Visitor	-		√
Iteretor	-	√	-
Ref. Count	-	-	√

TABLE I

LEVEL OF INTRUSION IN SOFTWARE

B. *Difficulties in Using Design Patterns*

The major difficulties in applying design patterns are as follows

- *Lack of standard implementations:* There are very few freely available robust implementations of design patterns in C++. Implementing robust and reliable design patterns such as Singleton, Visitor, Factory, Reference Counting etc are non-trivial task.
- *Design patterns are just software tool:* Design patterns are not part of a language. They are just some valuable software tricks, therefore they are likely to have different interpretations and implementation by various people. It is easy to create hard to understandable code which is against the very tenet of design patterns.
- *Breaking the hierarchy:* Existing applications might have to rearrange their hierarchies or use multiple inheritance, both are difficult and error prone. Language such as JAVA has advantages over C++ as it directly or indirect inherits every class from one superclass “Object” and support only single inheritance.
- *Powerful patterns need high intrusion:* Some of the design patterns such as Visitor and Observer patterns could realize their full potentials only when the user could change or reorganize the code substantially which may require lots of changes in the code and therefore the cost of reengineering could be higher.

V. CONCLUSIONS

Despite many shortcomings, legacy codes are too important to be left aside in the application development. Our experiments have shown that with design patterns and generic programming we can develop new systems which are very adaptable and extendable. We have applied GoF design pattern in our mesh generation application and we can categorically say that design pattern improve the system and make them flexible. This motivates us to explore patterns which could be useful in distributed parallel computing.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
[2] M. Jazayeri, R. Loos, and D. R. Musser, “Lecture notes in computer science 1776 : Generic programming.”
[3] J. C. K. Beck, “Industrial experience with design patterns.”
[4] D. L. Parnas, “Software aging,” in *Proceedings of 16th International Conference on Software Engineering*. Sorrento, Italy: IEEE, 16–21 May 1994, pp. 279–87.
[5] E. J. Chikofsky and J. H. C. II, “Reverse engineering and design recovery: A taxonomy,” *IEEE Software*, vol. 7, no. 1, pp. 13–17, Jan. 1990.
[6] C. Verheof, “Towards automated modification of legacy assets,” programming Research Group, University of Amsterdam.
[7] J. Brunekreef and B. Diertens, “Towards a user-controlled software renovation,” programming Research Group, University of Amsterdam.
[8] S. Meyers, *Effective C++: 50 Specific Ways to Improve Your Programs and Design*. Addison-Wesley, 1992.
[9] B. J. Cox, “Planning the software industrial revolution,” *IEEE Software*, vol. 7, no. 6, June 1990.
[10] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley, 2002.
[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern Oriented Software Architecture: A System of Patterns*. New York: John Wiley and Sons, 1996.
[12] N. E. Fenton, *Software Metrics: A Rigorous Approach*. London: Chapman and Hall, 1991.
[13] F. Rudolf, G. Juha, M. Laszlo, and P. Jukka, “Recognizing design patterns in C++ programs with the integration of columbis and maisa,” Department of Computer Science, Univ. of Helsinki, Tech. Rep., 2000.