

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

CS780 Project Presentation
Chaman Singh Verma
Ling Liu

12/15/2003

1

Outline

- Introduction
- Re-engineer a legacy system
- Evaluation and observations
- Conclusion

12/15/2003

2

Introduction

- Characteristics of legacy codes
 - Advantages
 - Evolve over a long period of time
 - Trustworthy in their limited functionality
 - User base is very high
 - Disadvantages
 - Conditional statements
 - Duplication of code for the same functionality
 - Lack of abstractions
 - Lack of separation of concerns
 - Monolithic interfaces

12/15/2003

3

Objectives

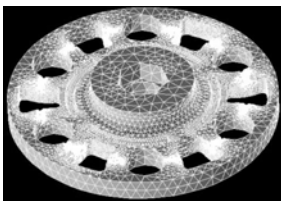
- Use design patterns to modify legacy code
- Explore the possibility of re-engineer a 3D Mesh Generation system using design patterns
- Research focus: Flexibility vs. Performance

12/15/2003

4

Applying Design Patterns in Mesh Generation

Carrier Image



<http://www.andrew.cmu.edu/user/sowen/mesh.html>
Meshing Research Corner

12/15/2003

5

Mesh Generation

- Define the shape
- Generate outer boundaries
- Create refinement boundaries
- Apply 3D Mesh

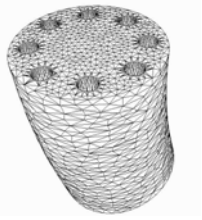


12/15/2003

6

Mesh Generation

Re-engineer the 3D
Mesh Generation
legacy code using
GoF patterns



Chaman's pipe

Gamma, Helm, Johnson, Vlassides

12/15/2003

7

Experimenting Design Patterns

- Adapter pattern
- Bridge pattern
- Factory method pattern
- Memento pattern
- Prototype pattern
- Observer pattern
- Singleton pattern
- Iterator pattern
- Template pattern
- State pattern
- Strategy pattern
- Visitor pattern

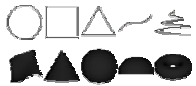
12/15/2003

8

Adapter pattern

■ Intent

- Convert the interface of a class into another interface clients expect
- Let classes with incompatible interface work together



■ Example:

- NURBS_Curve_t vs. Curve

12/15/2003

9

Adapter (continued)

■ NURBS_Curve_t class

```
class Curve{
    Public: Curve(NURBS_Curve_t *c);
    Point2D evaluate(double t);
    Point2D derivative(double t);
    Private: NURBS_Curve_t *curveold;
};
```

■ A glimpse of the old evaluate function

```
point_t pt = NURBS_EvalCurve(curveold, t);
```

12/15/2003

10

Adapter (continued)

■ A glimpse of the old evaluate function

```
point_t pt = NURBS_EvalCurve(olddcurve, t);
```

■ User expects to have a return value Point2D

```
Point2D evaluate(double t);
point_t pt = NURBS_EvalCurve(olddcurve, t);
Point2D result;
result[0] = pt.x;
result[1] = pt.y;
}
```

12/15/2003

11

Bridge pattern

■ Intent

- Decouple an interface from its implementation

```
class MeshGen2D {
public:
    MeshGen2D() { rep = new MeshGen2DImpl(); }
    void setData(int d)
    { rep->setData(d); }
    int getData() const {
    { return rep->getData(); }
private:
    MeshGen2DImpl *rep;
};
```

12/15/2003

12

Bridge (continued)

■ Example

- MeshGen2D.h
- May have MeshGen2DImp1, MeshGen2DImp2, ...

```
class MeshGen2DImp{
public:
    MeshGen2DImp();
    void setData( int d ) { data = d; }
    int getData() const { return data; }
private:
    int data;
};
```

12/15/2003

13

Factory Method

■ Intent

- Define an interface for creating an object
- Let subclasses decide which class to instantiate

12/15/2003

14

Factory (continued)

■ Consider the following code segment

```
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;
        }
    }
}
```

12/15/2003

15

Factory (continued)

■ Here is how our program will be

```
Factory<GeoEntity> factory;
factory.registerProduct( 0, GeoVertex::create);
factory.registerProduct( 1, GeoEdge::create);
factory.registerProduct( 2, GeoFace::create);
factory.registerProduct( 3, GeoCell::create);
```

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

12/15/2003

16

Factory (continued)

■ In Factory

```
template<class Product>
Product* Factory<Product> :: newProduct( int productID ){
    typename std::map<int,createCallback>::iterator iter;
    iter = callbacks.find(productID);
    if( iter == callbacks.end() ) {
        cout << " Unknown " << endl;
        throw std::runtime_error("unknown Shape ID"); }
    return (iter->second); }
```

■ Inside each class GeoCell, GeoVertex...

```
class GeoVertex{
public:
    static GeoVertex* create() {
        GeoVertex *gv = new GeoVertex();
        return gv;
    }
};
```

12/15/2003

17

Observer pattern

■ Intent:

- Define a one-to-many dependency between objects
- When one object changes state, all its dependents are notified and updated automatically

12/15/2003

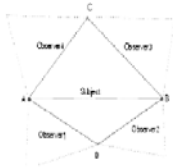
18

Observer (continued)

■ Example

Add observers of an edge AB.

```
edgeAB->attach( edgeCB );
edgeAB->attach( edgeAC );
edgeAB->attach( edgeAD );
edgeAB->attach( edgeDB );
```



12/15/2003

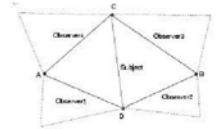
19

Observer (continued)

■ Example

During edge flipping,
notify all the observers
edgeAB->notify();

```
template <class T>
void Subject<T>::notify() {
    list<Observer*>::iterator i;
    for(i=observers.begin(); i!= observers.end(); i++)
        (*i)->Update();
}
```



12/15/2003

20

Visitor pattern

■ Intent

- Refine the functionality of the class without changing it

12/15/2003

21

Visitor (continued)

■ Example:

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

■ Change to:

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

12/15/2003

22

Visitor (continued)

■ Class Grid – add accept(v)

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

12/15/2003

23

Visitor (continued)

■ Each visitor looks like

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

■ Adding/refining function is easy

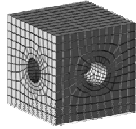
```
int main(){
    Grid2D g2d;
    Visitor<Face> *varea = new AreaVisitor;
    grid.accept(varea);
    Visitor<Face> *vasr = new AspectVisitor;
    grid.accept(vasr);
}
```

12/15/2003

24

Other patterns

- Iterator
- Template
- Prototype
- Memento
- Singleton
- State
- Strategy



12/15/2003

25

Iterator pattern

- Hide the details of container from user

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

12/15/2003

26

Iterator pattern

- When user uses the Iterator

```
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);
        .
        .
    }
}
```

12/15/2003

27

Template pattern

- Template Method defines an algorithm in terms of abstract operations that subclasses override to provide concrete behavior

```
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";}
};
```

12/15/2003

28

Prototype pattern

- Intent
 - Specify the kinds of objects to create
 - Using a prototype instance
 - Create new objects by copying this prototype

12/15/2003

29

Prototype

- Example

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

12/15/2003

30

Prototype

- Disadvantage:
 - Using a switch statement to identify the type of a parent object
 - Assuming a finite number of objects
 - Adding new type requires changing of code
- Using Prototype Pattern

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

Every class needs a clone function

12/15/2003

31

Memento pattern

- Intent
 - Helps remember the state of a class
 - When you want to roll back to a state
 - Memento helps you to set the state because it remembers it

12/15/2003

32

Singleton pattern

- Intent
 - Ensure a class only has one instance
 - Provide a global point of access to it

12/15/2003

33

Singleton: a glimpse of the code

```
template <class T>  
class Singleton  
{  
public:  
    // Returns the instance of Singleton  
    static T *instance ();  
    // Delete the pointer to the data  
    static void clear ();  
private:  
    static bool firstTime;  
    static T *pData;  
    Singleton ();  
    Singleton (const Singleton &);  
    Singleton & operator = (const Singleton &);  
};
```

12/15/2003

34

State pattern

- Intent
 - Allow an object to alter its behavior when its internal state changes
- Example

```
switch( cavityState){  
    case 0:goodCavity();break;  
    case 1:lockedCavity();break;  
    case 2:sliverCavity();break;  
    case 3:waitingCavity();break;  
}
```

12/15/2003

35

State pattern

- Replace switch statement by passing stateID to state-manager

```
int main(){  
    CavityState *cavityState;  
    cavityState->Register(1, new GoodCavity);  
    cavityState->Register(2, new LockeCavity);  
    cavityState->Register(3, new SliverCavity);  
    cavityState->Register(4, new WaitingCavity);  
    currentState = cavityState->getState(num);  
    currentState->Operation(num);  
}
```

12/15/2003

36

State Pattern

Each state:

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

12/15/2003

37

Strategy pattern

■ Intent

- Define a family of algorithms
- Algorithms can change independently

12/15/2003

38

Strategy (continued)

■ Example

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

Change to

```
int main(){
    MeshGen2D *meshGen;
    algRepository->Register(1, DelaunayMethod );
    algRepository->Register(2, AdvancedFrontMethod );
    currentStrategy = algRepository->getAlgorithm(1);
    meshGen->currentStrategy->applyAlgorithm(); }
```

12/15/2003

39

Evaluation

■ Flexibility vs. Performance

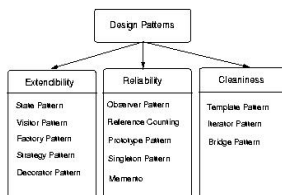
- Increased extensibility
- System more flexible
- Maintainability
- Performance: at most 5 percent longer
- Suitable for scientific computing

12/15/2003

40

Other observations

■ Characterization of the design patterns



12/15/2003

41

Difficulty in Using Design Patterns

- Lack of standard implementations
- Different interpretations
 - Example:
 - Visitor and Observer
- Powerful patterns need high intrusion

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

TABLE 1
LEVEL OF INTRUSION IN SOFTWARE

12/15/2003

42

Conclusion

- Re-engineered a 3D Mesh Generation Code
- Enhanced extensibility and flexibility
- Performance cost is marginal