Abstract Factory

Intent

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Problem

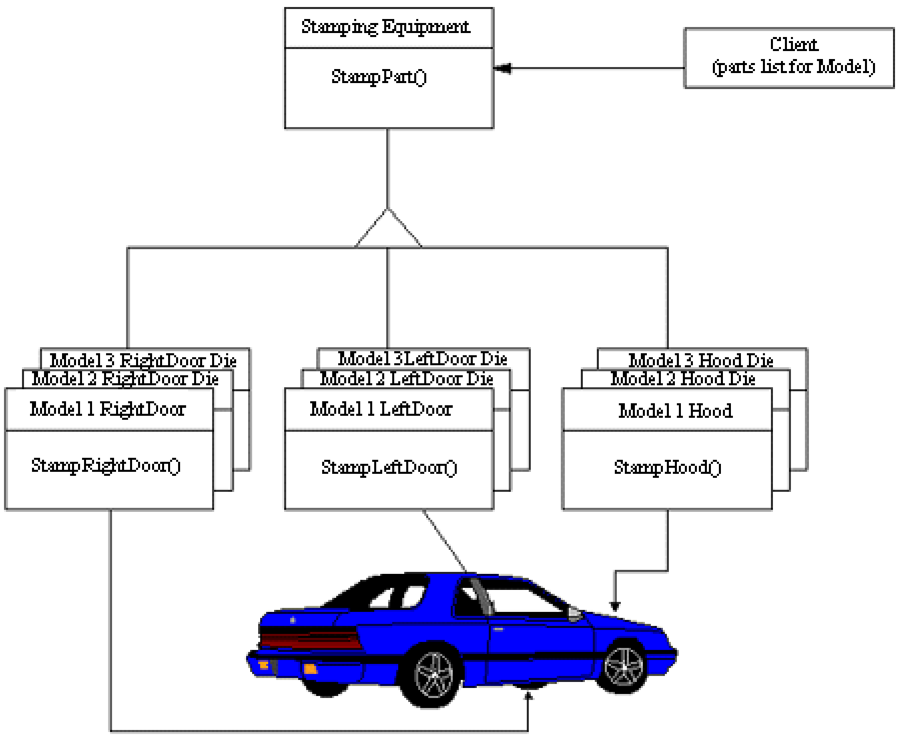
If an application is to be portable, it needs to encapsulate platform dependencies. These "platforms" might include: windowing system, operating system, database, etc. Too often, this encapsulation is not engineered in advance, and lots of #ifdef case statements with options for all currently supported platforms begin to procreate like rabbits throughout the code.

Discussion

Provide a level of indirection that abstracts the creation of families of related or dependent objects without directly specifying their concrete classes. The "factory" object has the responsibility for providing creation services for the entire platform family. Clients never create platform objects directly, they ask the factory to do that for them.

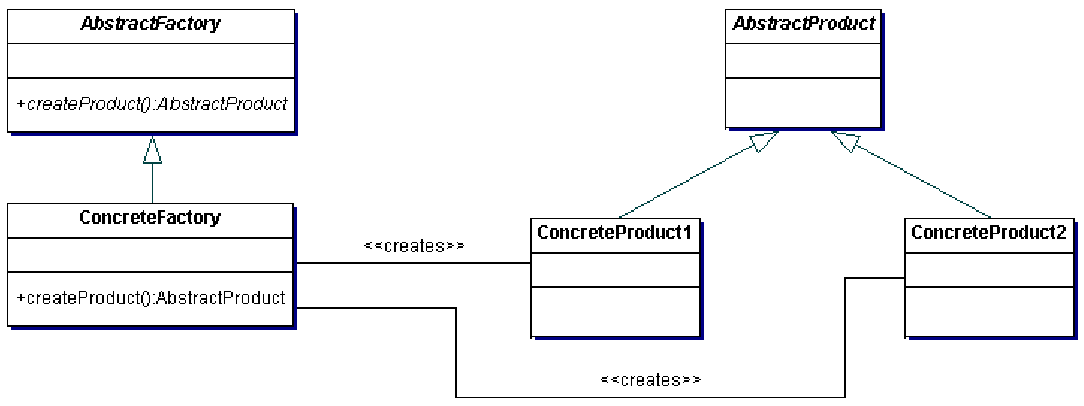
This mechanism makes exchanging product families easy because the specific class of the factory object appears only once in the application - where it is instantiated. The application can wholesale replace the entire family of products simply by instantiating a different concrete instance of the abstract factory.

Because the service provided by the factory object is so pervasive, it is routinely implemented as a Singleton.



Structure

The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes. This pattern is found in the sheet metal stamping equipment used in the manufacture of Japanese automobiles. The stamping equipment is an Abstract Factory which creates auto body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars. Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used profitably. At other times they are complementary: Abstract Factory might store a set of Prototypes from which to clone and return product objects [GOF, p126], Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementation. [GOF, pp81,134]

Abstract Factory, Builder, and Prototype define a factory object that's responsible for knowing and creating the class of product objects, and make it a parameter of the system. Abstract Factory has the factory object producing objects of several classes. Builder has the factory object building a complex product incrementally using a correspondingly complex protocol. Prototype has the factory object (aka prototype) building a product by copying a prototype object. [GOF, p135]

Abstract Factory classes are often implemented with Factory Methods, but they can also be implemented using Prototype. [GOF, p95]

Abstract Factory can be used as an alternative to Facade to hide platform-specific classes. [GOF, p193]

Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately. [GOF, p105]

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. [GOF, p136]

PROGRAM 1:

```
// Purpose.  Abstract Factory design pattern demo.
//
// Discussion.  "Think of constructors as factories that churn out objects".
// Here we are allocating the constructor responsibility to a factory object,
// and then using inheritance and virtual member functions to provide a
// "virtual constructor" capability.  So there are two dimensions of
// decoupling occurring.  The client uses the factory object instead of "new"
// to request instances; and, the client "hard-wires" the family, or class,
// of that factory only once, and throughout the remainder of the
// application only relies on the abstract base class.

#include <iostream.h>

class Shape
{
public:
   Shape()             { id_ = total_++; }
   virtual void draw()  = 0;
protected:
   int         id_;
   static int  total_;
};
int Shape::total_ = 0;

class Circle : public Shape
{
 public:
   void draw() { cout << "circle " << id_ << ": draw" << endl; }
 };
class Square : public Shape
{
 public:
   void draw() { cout << "square " << id_ << ": draw" << endl; }
 };
class Ellipse : public Shape
{
public:
   void draw() { cout << "ellipse " << id_ << ": draw" << endl; }
};
class Rectangle : public Shape
{
public:
   void draw() { cout << "rectangle " << id_ << ": draw" << endl; }
};
```

```
class Factory
{
public:
   virtual Shape* createCurvedInstance()   = 0;
   virtual Shape* createStraightInstance() = 0;
};
class SimpleShapeFactory : public Factory
{
public:
   Shape* createCurvedInstance()   { return new Circle; }
   Shape* createStraightInstance() { return new Square; }
};
class RobustShapeFactory : public Factory
{
public:
   Shape* createCurvedInstance()   { return new Ellipse; }
   Shape* createStraightInstance() { return new Rectangle; }
};

void main()
{
#ifdef SIMPLE
   Factory*  factory = new SimpleShapeFactory;
#elif ROBUST
   Factory*  factory = new RobustShapeFactory;
#endif
   Shape*    shapes[3];

   shapes[0] = factory->createCurvedInstance();   // shapes[0] = new Ellipse;
   shapes[1] = factory->createStraightInstance(); // shapes[1] = new Rectangle;
   shapes[2] = factory->createCurvedInstance();   // shapes[2] = new Ellipse;

   for (int i=0; i < 3; i++)
      shapes[i]->draw();
}

// ellipse 0: draw
// rectangle 1: draw
// ellipse 2: draw
```

PROGRAM 2:

```
// Purpose.  Abstract Factory Discussion.  Trying to maintain
// portability across multiple "platforms" routinely requires lots of
// preprocessor "case" stmts.  The Factory pattern suggests defining
// a creation services interface in a Factory base class, and
// implementing each "platform" in a separate Factory derived class.

#include <iostream.h>

class Widget
{
public:

        virtual void draw() = 0;
```

```
};

class MotifBtn : public Widget
{
public:
          void draw()
          {
           cout << "MotifBtn" << endl;
          }
};

class WindowsBtn : public Widget
{
public:
          void draw()
          {
            cout << "WindowsBtn"<< endl;
          }
};

void doThisWindow()
{
// create window, attach btn
          #ifdef MOTIF
                    Widget* w = new MotifBtn;
          #else // WINDOWS
                    Widget* w = new WindowsBtn;
          #endif
          w->draw();
 }

void doThatWindow()
{
// create window, attach btn
#ifdef MOTIF
          Widget* w = new MotifBtn;
#else // WINDOWS
          Widget* w = new WindowsBtn;
#endif
          w->draw();
 }

void main( void )
{
// create window, attach btn
#ifdef MOTIF
          Widget* w = new MotifBtn;
#else // WINDOWS
          Widget* w = new WindowsBtn;
#endif

w->draw();

doThisWindow();
doThatWindow();
```

```
}

// WindowsBtn
// WindowsBtn
// WindowsBtn
```

PROGRAM 3:

```
#include <iostream.h>

class Widget
{
public:
	virtual void draw() = 0;
};

class MotifBtn : public Widget
{
public:
	void draw() { cout << "MotifBtn"<< endl; }
};

class WindowsBtn : public Widget
{
public:
	void draw() { cout << "WindowsBtn"<< endl; }
};

class Factory
{
public:
virtual Widget* createBtn() = 0;
};

class MotifFactory : public Factory
{
public:
	Widget* createBtn() {return new MotifBtn; }
};

class WindowsFactory : public Factory
{
public:
	Widget* createBtn() {return new WindowsBtn; }
};

Factory* factory;

void doThisWindow()
{
// create window, attach btn
	Widget* w = factory->createBtn();
	w->draw();
}
```

```
void doThatWindow()
{
// create window, attach btn
        Widget* w = factory->createBtn();
        w->draw();
}

void main( void )
{
#ifdef MOTIF
        factory = new MotifFactory;
#else // WINDOWS
        factory = new WindowsFactory;
#endif

// create window, attach btn
        Widget* w = factory->createBtn();
        w->draw();

doThisWindow();
doThatWindow();
}

// MotifBtn
// MotifBtn
// MotifBtn
```

Builder

Intent

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Problem

An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.

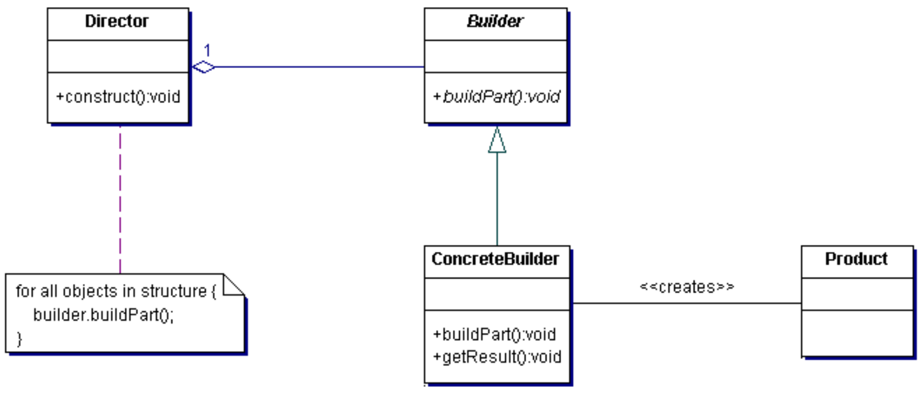
Discussion

Separate the algorithm for interpreting (i.e. reading and parsing) a stored persistence mechanism (e.g. RTF files) from the algorithm for building and representing one of many target products (e.g. ASCII, TeX, text widget). The focus/distinction is on creating complex aggregates.

The "director" invokes "builder" services as it interprets the external format. The "builder" creates part of the complex object each time it is called and maintains all intermediate state. When the product is finished, the client retrieves the result from the "builder".

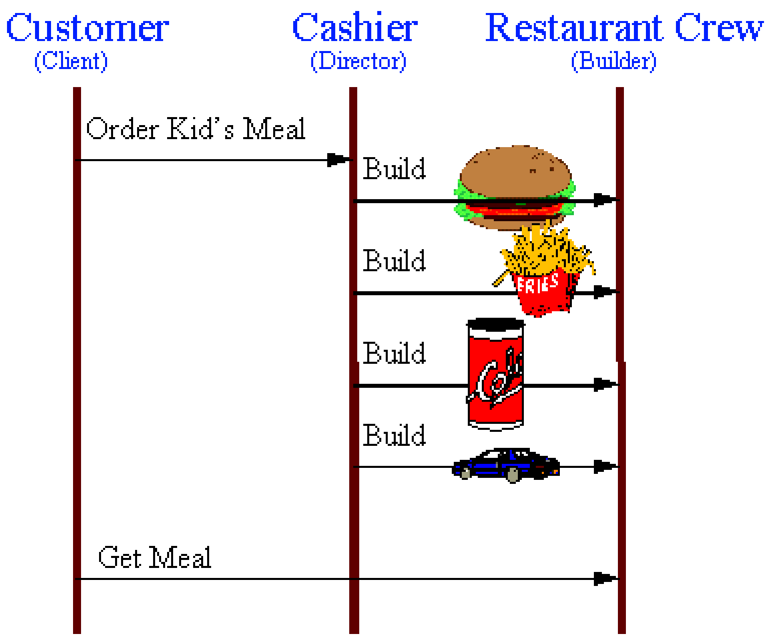
Affords finer control over the construction process. Unlike creational patterns that construct products in one shot, the Builder pattern constructs the product step by step under the control of the "director".

Structure



Example

The Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations. This pattern is used by fast food restaurants to construct children's meals. Children's meals typically consist of a main item, a side item, a drink, and a toy (e.g., a hamburger, fries, Coke, and toy car). Note that there can be variation in the content of the children's meal, but the construction process is the same. Whether a customer orders a hamburger, cheeseburger, or chicken, the process is the same. The employee at the counter directs the crew to assemble a main item, side item, and toy. These items are then placed in a bag. The drink is placed in a cup and remains outside of the bag. This same process is used at competing restaurants. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

Sometimes-creational patterns are complementary: Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementations. [GOF, p81, 134]

Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately. [GOF, p105]

Builder is to creation as Strategy is to algorithm. [Icon, p8-13]

Builder often builds a Composite. [GOF, p106]

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. [GOF, p136]

PROGRAM 1:

```
// Purpose.  Builder design pattern demo.
//
// Discussion.  The forte of Builder is constructing a complex object step
// by step.  An abstract base class declares the standard construction
// process, and concrete derived classes define the appropriate
// implementation for each step of the process.  In this example,
// "distributed work packages" have been abstracted to be persistent and
// platform independent.  This means that the platform-specific mechanism
// for implementing files, queues, and concurrency pathways is defined in
// each platform's concrete derived class.  A single "reader" object (i.e.
// parser) retrieves the archived specification for a DistrWorkPackage and
// proceeds to delegate each build step to the builder object that was
// registered by the client.  Upon completion, the client retrieves the
// end result from the builder.

#include <iostream.h>
#include <stdio.h>
#include <string.h>

enum PersistenceType { File, Queue, Pathway };

struct PersistenceAttribute
{
		PersistenceType  type;
		char value[30];
};

class DistrWorkPackage
{
public:
DistrWorkPackage( char* type )
{
		sprintf( _desc, "Distributed Work Package for: %s", type);
}
void setFile( char* f, char* v )
{
    sprintf( _temp, "\n  File(%s): %s", f, v );
    strcat( _desc, _temp);
}
void setQueue( char* q, char* v )
{
		sprintf( _temp, "\n  Queue(%s): %s", q, v );
		strcat( _desc, _temp);
}
```

```
void setPathway( char* p, char* v )
{
        sprintf( _temp, "\n  Pathway(%s): %s", p, v );
        strcat(_desc,_temp);
}
const char* getState() { return _desc; }
private:
        char  _desc[200], _temp[80];
};

class Builder
{
public:
        virtual void configureFile( char* )    = 0;
        virtual void configureQueue( char* )   = 0;
        virtual void configurePathway( char* ) = 0;
        DistrWorkPackage* getResult() { return _result; }
protected:
        DistrWorkPackage*  _result;
};

class UnixBuilder : public Builder
{
public:
        UnixBuilder() { _result = new DistrWorkPackage( "Unix" ); }

        void configureFile( char* name )
        {
                _result->setFile( "flatFile", name );
        }
        void configureQueue( char* queue )
        {
                _result->setQueue( "FIFO", queue );
        }
        void configurePathway( char* type )
        {
                _result->setPathway( "thread", type );
        }
};

class VmsBuilder : public Builder
{
public:
        VmsBuilder() { _result = new DistrWorkPackage( "Vms" ); }
        void configureFile( char* name )
        {
                _result->setFile( "ISAM", name );
        }
        void configureQueue( char* queue )
        {
                _result->setQueue( "priority", queue );
        }
        void configurePathway( char* type )
        {
                _result->setPathway( "LWP", type );
        }
};

class Reader
{
public:
        void setBuilder( Builder* b ) { _builder = b; }
        void construct( PersistenceAttribute[], int );
private:
        Builder* _builder;
};

void Reader::construct( PersistenceAttribute list[], int num )
{
        for (int i=0; i < num; i++)
                if (list[i].type == File)
                        _builder->configureFile( list[i].value );
                else if (list[i].type == Queue)
                        _builder->configureQueue( list[i].value );
                else if (list[i].type == Pathway)
                        _builder->configurePathway( list[i].value );
}

const int  NUM_ENTRIES = 6;
PersistenceAttribute  input[NUM_ENTRIES] = { {File, "state.dat"},
        {File,"config.sys"}, {Queue, "compute"}, {Queue, "log"},
        {Pathway, "authentication"}, {Pathway, "error processing"} };

void main()
{
        UnixBuilder  unixBuilder;
        VmsBuilder   vmsBuilder;
        Reader       reader;

        reader.setBuilder( &unixBuilder );
        reader.construct( input, NUM_ENTRIES );
        cout << unixBuilder.getResult()->getState() << endl;

        reader.setBuilder( &vmsBuilder );
        reader.construct( input, NUM_ENTRIES );
        cout << vmsBuilder.getResult()->getState() << endl;
}

// Distributed Work Package for: Unix
//   File(flatFile): state.dat
//   File(flatFile): config.sys
//   Queue(FIFO): compute
//   Queue(FIFO): log
//   Pathway(thread): authentication
//   Pathway(thread): error processing
//   Distributed Work Package for: Vms
//   File(ISAM): state.dat
//   File(ISAM): config.sys
//   Queue(priority): compute
//   Queue(priority): log
//   Pathway(LWP): authentication
//   Pathway(LWP): error processing
```

PROGRAM 2:

```
// Purpose.  Builder The monolithic design supports a
// single representation.  The Builder design allows a different rep per
// Builder derived class, and the common input and parsing have been
//de-fined in the Director class.  The D constructs, the B returns result.

class Array
{
public:
        void addFront( char ch )
        {
                lst.push_front( ch );
        }
        void addBack( char ch )
        {
                lst.push_back( ch );
        }
        void traverse()
        {
                for (i=0; i < lst.size(); i++)
                cout << lst[i] << ' ';
                cout << endl;
        }

private:
        deque<char> lst;   int i;
};

string in[] = { "fa", "bb", "fc","bd", "fe", "bf", "fg", "bh" };

void main( void )
{
        Array  list;
        for (int i=0; i < 8; i++)
                if (in[i][0] == 'f')
                        list.addFront( in[i][1] );
                else if (in[i][0] == 'b')
                        list.addBack( in[i][1] );
        list.traverse();
}

// g e c a b d f h
```

```
//////////////////\\\\\\\\\\\\\\\\\\

class Array
{
 public:
        virtual void traverse() = 0;
};

class OneEnded : public Array
{
public:
        friend class BuilderOne;
        void traverse()
        {
                for (i=0; i < lst.size(); i++)
                cout << lst[i] << ' ';
                cout << endl;
        }
private:
vector<char> lst;   int i;
};

class TwoEnded : public Array
{
public:
        friend class BuilderTwo;
        void traverse()
        {
                for (i=0; i < lst.size(); i++)
                        cout << lst[i] << ' ';
                cout << endl;
        }
private:
        deque<char> lst;   int i;
};
```

PROGRAM 3:

```
class Builder
{
public:
        virtual void addFront( char ) = 0;
        virtual void addBack( char ) = 0;
        virtual Array& getResult()  = 0;
};
class BuilderOne : public Builder
{
public:
        void addFront( char ch )
        {
                one.lst.push_back( ch );
        }
        void addBack( char ch )
        {
                one.lst.push_back( ch );
        }
```

```
          Array& getResult()
          {
                    return one;
          }
private:
          OneEnded one;
};

class BuilderTwo : public Builder
{
public:
          void addFront( char ch ) {two.lst.push_front( ch ); }
          void addBack( char ch ) {two.lst.push_back( ch ); }
          Array& getResult() {return two; }
private:
          TwoEnded two;
};

string in[] = { "fa", "bb", "fc","bd", "fe", "bf", "fg", "bh" };

class Director
{
public:
          Director( Builder* b ) { setBuilder( b ); }
          void setBuilder( Builder* b ) {bldr = b; }
          void construct()
          {
                    for (int i=0; i < 8; i++)
                    if (in[i][0] == 'f')
                              bldr->addFront(in[i][1]);
                    else if (in[i][0] == 'b')
                              bldr->addBack(in[i][1]);
          }
private:
          Builder* bldr;
};

void main( void )
{
          BuilderOne one;
          BuilderTwo two;
          Director dir( &one );

          dir.construct();
          one.getResult().traverse();
          dir.setBuilder( &two );
          dir.construct();
          two.getResult().traverse();
}

// a b c d e f g h
// g e c a b d f h
```

Factory Method

Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Problem

A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

Discussion

Factory Method is to creating objects as Template Method is to implementing an algorithm. A superclass specifies all standard and generic behavior (using pure virtual "placeholders" for creation steps), and then delegates the creation details to subclasses that are supplied by the client.

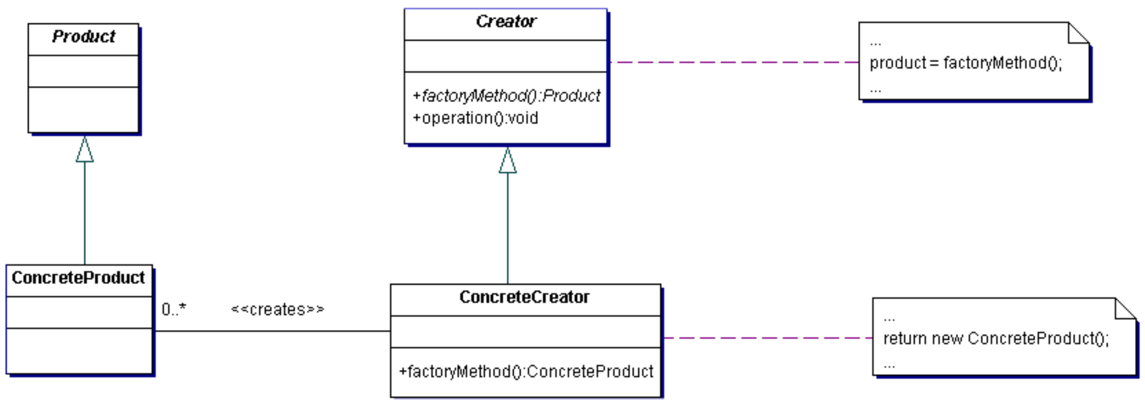
Factory Method makes a design more customizable and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation. [GOF, p136]

People often use Factory Method as the standard way to create objects; but it isn't necessary if: the class that's instantiated never changes, or instantiation takes place in an operation that subclasses can easily override (such as an initialization operation). [GOF, p136]

Factory Method is similar to Abstract Factory but without the emphasis on families.

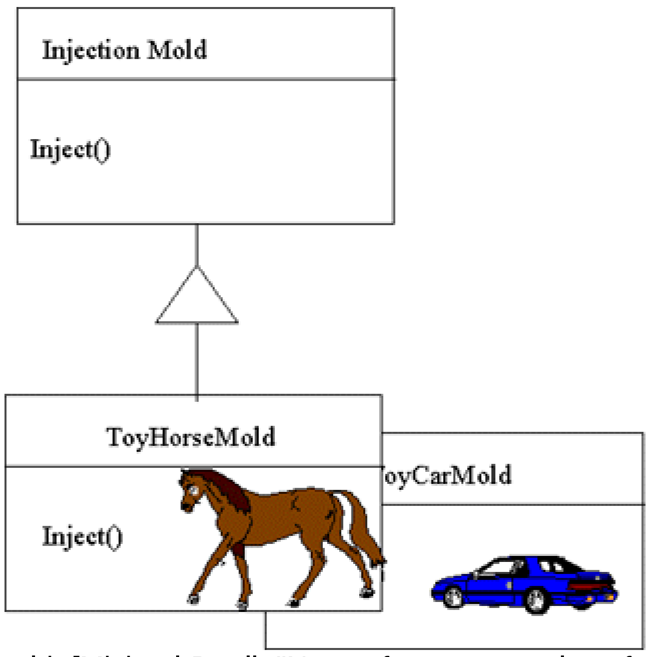
Factory Methods are routinely specified by an architectural framework, and then implemented by the user of the framework.

Structure



Example

The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is



determined by the mold. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]

Rules of thumb

Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype. [GOF, p95]

Factory Methods are usually called within Template Methods. [GOF, p116]

Factory Method: creation through inheritance. Prototype: creation through delegation.

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. [GOF, p136]

Prototype doesn't require subclassing, but it does require an Initialize operation. Factory Method requires subclassing, but doesn't require Initialize. [GOF, p116]

<u>Program Examples</u>

Example 1:

PROGRAM 1:

```
// Purpose.  Factory Method design pattern demo.
//
// Discussion.  Frameworks are applications (or subsystems) with "holes"
// in them.  Each framework specifies the infrastructure, superstructure,
// and flow of control for its "domain", and the client of the framework
// may: exercise the framework's default behavior "as is", extend selected
// pieces of the framework, or replace selected pieces.  The Factory
// Method pattern addresses the notion of "creation" in the context of
// frameworks.  In this example, the framework knows WHEN a new document
// should be created, not WHAT kind of Document to create.  The
// "placeholder" Application::CreateDocument() has been declared by the
// framework, and the client is expected to "fill in the blank" for
// his/her specific document(s).  Then, when the client asks for
// Application::NewDocument(), the framework will subsequently call the
// client's MyApplication::CreateDocument().

#include <iostream.h>

/* Abstract base class declared by framework */
class Document
{
public:
        Document( char* fn ) { strcpy( name, fn ); }
        virtual void Open()  = 0;
        virtual void Close() = 0;
        char* GetName()     { return name; }
private:
        char  name[20];
};

/* Concrete derived class defined by client */
class MyDocument : public Document
{
public:
        MyDocument( char* fn ) : Document(fn) { }
        void Open() { cout << "   MyDocument: Open()" << endl;}
        void Close(){ cout << "   MyDocument: Close()" << endl;}
};


/* Framework declaration */
class Application
{
public:
        Application() : _index(0)
        { cout << "Application: ctor" << endl; }

/* The client will call this "entry point" of the framework */
        NewDocument( char* name )
         {
                cout << "Application: NewDocument()" << endl;
// Framework calls the "hole" reserved for client
// customization
                _docs[_index] = CreateDocument( name );
                _docs[_index++]->Open();
         }

        OpenDocument()  { }
        void ReportDocs();


/* Framework declares a "hole" for the client to customize */
        virtual Document* CreateDocument( char* ) = 0;
private:
        int _index;
        /* Framework uses Document's base class */
        Document*  _docs[10];
};

void Application::ReportDocs()
{
        cout << "Application: ReportDocs()" << endl;
        for (int i=0; i < _index; i++)
                cout << "   " << _docs[i]->GetName() << endl;
}
```

```
/* Customization of framework defined by client */
class MyApplication : public Application
{
public:
        MyApplication() { cout << "MyApplication: ctor" << endl;}
        /* Client defines Framework's "hole" */
        Document* CreateDocument( char* fn )
        {
                cout << "   MyApplication: CreateDocument()" << endl;
                return new MyDocument( fn );
        }
};

void main()
 {
        /* Client's customization of the Framework */
        MyApplication  myApp;

        myApp.NewDocument( "foo" );
        myApp.NewDocument( "bar" );
        myApp.ReportDocs();
}

// Application: ctor
// MyApplication: ctor
// Application: NewDocument()
//    MyApplication: CreateDocument()
//    MyDocument: Open()
// Application: NewDocument()
//    MyApplication: CreateDocument()
//    MyDocument: Open()
// Application: ReportDocs()
//    foo
//    bar
```

PROGRAM 2:

```
// Purpose.  Factory Method creation via inheritance
// Discussion.  The architect has done an admirable job of
// decoupling the client from Stooge concrete derived
// classes, and, exercising polymor-phism.  But there remains
// coupling where instances are actually created.  If we design
// an "extra level of indirection" (a "factory method") and
// have clients use it (instead of "new"), then the last
// bit of coupling goes away.  The "factory method" (aka "virtual
// constructor") can be defined in the Stooge base class, or, in a
// separate "factory" class.  Note that main() is no longer
// dependent on Stooge derived classes.

#include <iostream>
using namespace std;

class Stooge
{
 public:
	virtual void slapStick() = 0;
};

class Larry : public Stooge
{
 public:
	void slapStick() {cout << "Larry: poke eyes" << endl; }
};
class Moe : public Stooge
{
 public:
	void slapStick() {cout << "Moe: slap head" << endl; }
};
class Curly : public Stooge
{
public:
	void slapStick() {cout << "Curly: suffer abuse" << endl; }
};

void main( void )
{
	Stooge*  roles[10];
	int      in, j, i = 0;

	cout << "L(1) M(2) C(3) Go(0): ";
	cin >> in;
while (in)
	{
	if (in == 1)
		roles[i++] = new Larry;
	else if (in == 2)
		roles[i++] = new Moe;
	else
		roles[i++] = new Curly;
```

```
        cout << "L(1) M(2) C(3) Go(0): ";
        cin >> in;
        }
for (j=0; j < i; j++)
roles[j]->slapStick();
for (j=0; j < i; j++)
delete roles[j];
}
```

PROGRAM 3:

```
#include <iostream>
using namespace std;

class Stooge
{
public:
        virtual void slapStick() = 0;
};

class Factory
{
public:
        // Factory Method (virtual ctor)
        static Stooge* create( int );
};

void main( void )
{
Stooge*  roles[10];
int     in, j, i = 0;

while (1)
{
        cout << "L(1) M(2) C(3) Go(0): ";
        cin >> in;
        if ( ! in ) break;
        roles[i++] = Factory::create(in);
}
for (j=0; j < i; j++)
roles[j]->slapStick();
for (j=0; j < i; j++)
delete roles[j];
}

class Larry : public Stooge
{
public:
        void slapStick() {cout << "Larry: poke eyes"<< endl; }
};
class Moe : public Stooge
{
 public:
        void slapStick() {cout << "Moe: slap head"<< endl; }
};
```

```
class Curly : public Stooge
{
 public:
        void slapStick() {cout << "Curly: suffer abuse"<< endl; }
};

Stooge* Factory::create( int in )
{
        if (in == 1)
                return new Larry;
        else if (in == 2)
                return new Moe;
        else
                return new Curly;
}

// L(1) M(2) C(3) Go(0): 1
// L(1) M(2) C(3) Go(0): 2
// L(1) M(2) C(3) Go(0): 3
// L(1) M(2) C(3) Go(0): 1
// L(1) M(2) C(3) Go(0): 0
// Larry: poke eyes
// Moe: slap head
// Curly: suffer abuse
// Larry: poke eyes
```

Prototype

Intent

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Problem

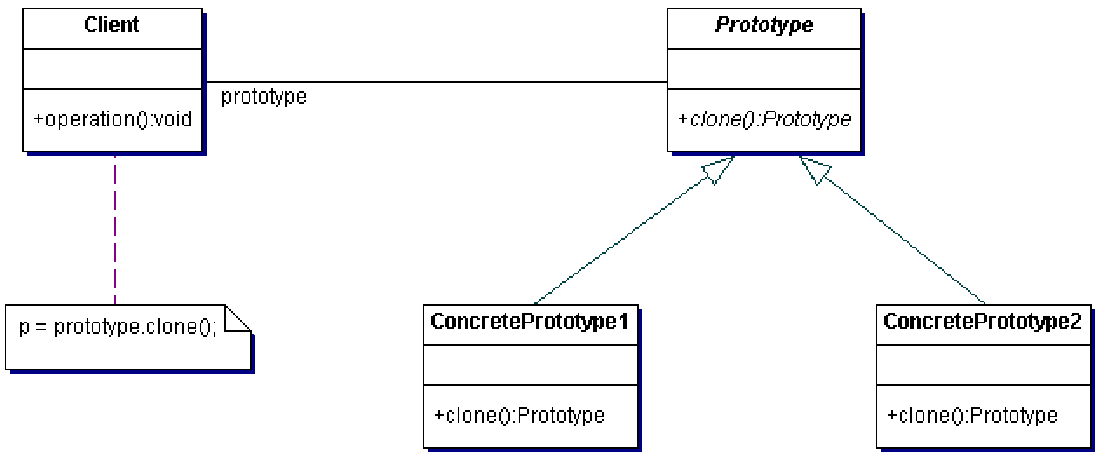
Application "hard wires" the class of object to create in each "new" expression.

Discussion

Declare an abstract base class that specifies a pure virtual "clone" method, and, maintains a dictionary of all "cloneable" concrete derived classes. Any class that needs a "polymorphic constructor" capability: derives itself from the abstract base class, registers its prototypical instance, and implements the clone() operation.

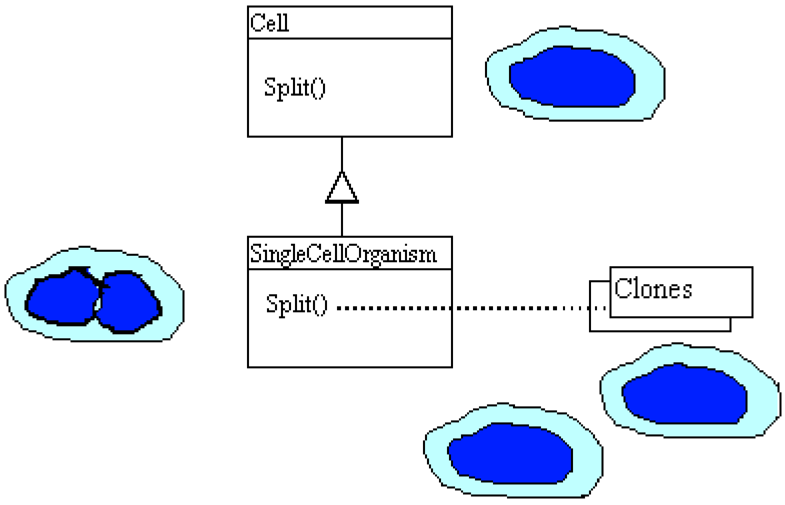
The client then, instead of writing code that invokes the "new" operator on a hard-wired class name, calls a "clone" operation on the abstract base class, supplying a string or enumerated data type that designates the particular concrete derived class desired.

Structure



Example

The Prototype pattern specifies the kind of objects to create using a prototypical instance. Prototypes of new products are often built prior to full production, but in this example, the prototype is passive and does not participate in copying itself. The mitotic division of a cell - resulting in two identical cells - is an example of a prototype that plays an active role in copying itself and thus, demonstrates the Prototype pattern. When a cell splits, two cells of identical genotvpe result. In other words, the cell clones itself. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used properly. At other times they are complementary: Abstract Factory might store a set of Prototypes from which to clone and return product objects [GOF, p126]. Abstract Factory, Builder, and Prototype can use Singleton in their implementations. [GOF, p81, 134].

Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype. [GOF, p95]

Factory Method: creation through inheritance. Protoype: creation through delegation.

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Protoype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. [GOF, p136]

Prototype doesn't require subclassing, but it does require an "initialize" operation. Factory Method requires subclassing, but doesn't require Initialize. [GOF, p116]

Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well. [GOF, p126]

Program Examples

Example 1:

PROGRAM 1:

```
// Purpose.  Prototype  creation via delegation
// Discussion.  The architect has done an admirable job
// of decoupling the client from Stooge concrete derived
// classes and exercising polymorphism.  But there remains
// coupling where instances are actually created.
// If we design an "extra level of indirection" (a "factory")
// and have clients use it (instead of "new"), then the last bit
// of coupling goes away.  The Prototype pattern suggests
// delegating the creation service to contained objects that
// know how to "clone" themselves.  This strategy also allows
// us to retire the "case" statement in main().

#include <iostream.h>

class Stooge
{
public:
virtual void slapStick() = 0;
};

class Larry : public Stooge
{
public:
void slapStick() {cout << "Larry: poke eyes"<< endl; }
};

class Moe : public Stooge
{
 public:
        void slapStick() {cout << "Moe: slap head" << endl; }
};
```

```
class Curly : public Stooge
{
 public:
	void slapStick(){cout << "Curly: suffer abuse"<< endl; }
};

void main( void )
{
Stooge*  roles[10];
int      in, j, i = 0;

cout << "L(1) M(2) C(3) Go(0): ";
cin >> in;
while (in)
	 {
	if (in == 1)
		roles[i++] = new Larry;
	else if (in == 2)
		roles[i++] = new Moe;
	else
		roles[i++] = new Curly;
	cout << "L(1) M(2) C(3) Go(0): ";
	cin >> in;
	}

for (j=0; j < i; j++)
roles[j]->slapStick();

for (j=0; j < i; j++)
delete roles[j];
}
```

PROGRAM 2:

```
#include <iostream.h>

class Stooge
{
 public:
	virtual Stooge* clone() = 0;
	virtual void slapStick() = 0;
};

class Factory
{
public:
	static Stooge* create( int i );
private:
	static Stooge* prototypes_[4];
};
```

```
void main( void )
{
Stooge*  roles[10];
int      in, j, i = 0;

cout << "L(1) M(2) C(3) Go(0): ";
cin >> in;
while (in)
        {
                roles[i++] = Factory::create(in);
                cout << "L(1) M(2) C(3) Go(0): ";
                cin >> in;
        }

for (j=0; j < i; j++)
roles[j]->slapStick();

for (j=0; j < i; j++)
delete roles[j];
}

//////////////////////////////////
class Larry : public Stooge
{
 public:
        Stooge* clone() { return new Larry; }
        void slapStick() {cout << "Larry: poke eyes" << endl; }
};

class Moe : public Stooge
{
 public:
        Stooge* clone() { return new Moe; }
        void slapStick() {cout << "Moe: slap head" << endl; }
};

class Curly : public Stooge
{
 public:
        Stooge* clone() {return new Curly; }
        void slapStick() {cout << "Curly: suffer abuse" << endl; }
};

Stooge* Factory::prototypes_[] = { 0,new Larry, new Moe, new Curly };
Stooge* Factory::create( int i ) {return prototypes_[i]->clone(); }

// L(1) M(2) C(3) Go(0): 1
// L(1) M(2) C(3) Go(0): 2
// L(1) M(2) C(3) Go(0): 3
// L(1) M(2) C(3) Go(0): 1
// L(1) M(2) C(3) Go(0): 0
// Larry: poke eyes
// Moe: slap head
// Curly: suffer abuse
// Larry: poke eyes
```

PROGRAM 3:

```
// Purpose.  Prototype design pattern demo
//
// Discussion.  Image base class provides the mechanism for storing,
// finding, and cloning the prototype for all derived classes.  Each
// derived class specifies a private static data member whose
// initialization "registers" a prototype of itself with the base class.
// When the client asks for a "clone" of a certain type, the base class
// finds the prototype and calls clone() on the correct derived class.

#include <iostream.h>

enum imageType { LSAT, SPOT };


class Image
{
public:
	virtual void draw()= 0;
	static  Image* findAndClone( imageType );
protected:
	virtual imageType returnType() = 0;
	virtual Image*    clone()      = 0;
	// As each subclass of Image is declared, it registers its prototype
	static void addPrototype( Image* image )
		{ _prototypes[_nextSlot++] = image; }
private:
	// addPrototype() saves each registered prototype here
	static Image* _prototypes[10];
	static int    _nextSlot;
};

Image* Image::_prototypes[];
int    Image::_nextSlot;

// Client calls this public static member function when it needs an instance
// of an Image subclass
Image* Image::findAndClone( imageType type )
{
	for (int i=0; i < _nextSlot; i++)
		if (_prototypes[i]->returnType() == type)
			return _prototypes[i]->clone();
}

class LandSatImage : public Image
{
public:
	imageType returnType() { return LSAT; }
	void draw(){ cout << "LandSatImage::draw " << _id << endl; }
	// When clone() is called, call the one-argument ctor with a dummy arg
	Image* clone(){ return new LandSatImage( 1 ); }
```

```
protected:
        // This is only called from clone()
        LandSatImage(int dummy) { _id = _count++; }
private:
        // Mechanism for initializing an Image subclass - this causes the
        // default ctor to be called, which registers the subclass's prototype
        static LandSatImage _landSatImage;
        // This is only called when the private static data member is inited
        LandSatImage() { addPrototype( this ); }
        // Nominal "state" per instance mechanism
        int       _id;
        static int _count;
};

// Register the subclass's prototype
LandSatImage LandSatImage::_landSatImage;
// Initialize the "state" per instance mechanism
int LandSatImage::_count = 1;


class SpotImage : public Image
{
public:
        imageType returnType() { return SPOT; }
        void draw(){ cout << "SpotImage::draw " << _id << endl; }
        Image* clone(){ return new SpotImage( 1 ); }
protected:
        SpotImage( int dummy ) { _id = _count++; }
private:
        SpotImage() { addPrototype( this ); }
        static SpotImage _spotImage;
        int       _id;
        static int _count;
};

SpotImage SpotImage::_spotImage;
int      SpotImage::_count = 1;


// Simulated stream of creation requests
const int  NUM_IMAGES = 8;
imageType  input[NUM_IMAGES] =
        { LSAT, LSAT, LSAT, SPOT, LSAT, SPOT, SPOT, LSAT };




void main()
{
        Image*  images[NUM_IMAGES];

        // Given an image type, find the right prototype, and return a clone
        for (int i=0; i < NUM_IMAGES; i++)
                images[i] = Image::findAndClone( input[i] );
        // Demonstrate that correct image objects have been cloned
        for (i=0; i < NUM_IMAGES; i++)
            images[i]->draw();

        // Free the dynamic memory
        for (i=0; i < NUM_IMAGES; i++)
            delete images[i];
    }

    // LandSatImage::draw 1
    // LandSatImage::draw 2
    // LandSatImage::draw 3
    // SpotImage::draw 1
    // LandSatImage::draw 4
    // SpotImage::draw 2
    // SpotImage::draw 3
    // LandSatImage::draw 5
```

Singleton

Intent

Ensure a class has only one instance, and provide a global point of access to it.

Problem

Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Discussion

Make the class of the single instance object responsible for creation, initialization, access, and enforcement. Declare the instance as a private static data member. Provide a public static member function that encapsulates all initialization code, and provides access to the instance.

The client calls the accessor function (using the class name and scope resolution operator) whenever a reference to the single instance is required.

Singleton should be considered only if all three of the following criteria are satisfied:

- Ownership of the single instance cannot be reasonably assigned
- Lazy initialization is desirable
- Global access is not otherwise provided for

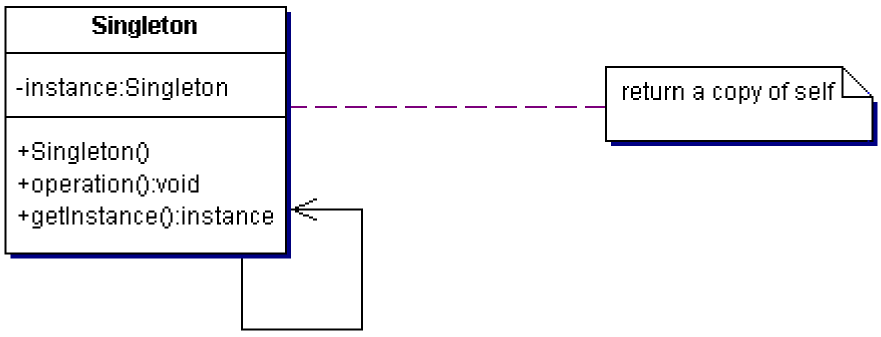
If ownership of the single instance, when and how initialization occurs, and global access are not issues, Singleton is not sufficiently interesting.

The Singleton pattern can be extended to support access to an application-specific number of instances.

The "static member function accessor" approach will not support subclassing of the Singleton class. If subclassing is desired, refer to the discussion in the book.

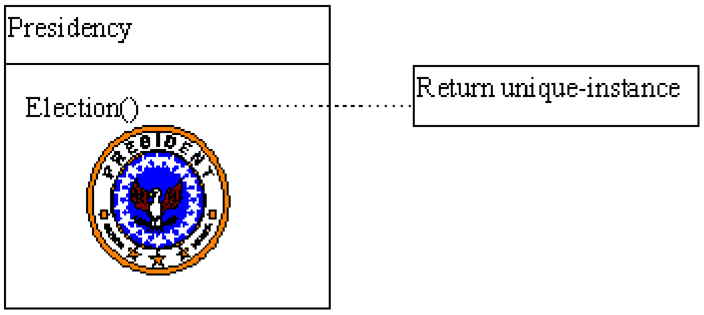
Deleting a Singleton class/instance is a non-trivial design problem. See "To kill a Singleton" by John Vlissides (*C++ Report*, Jun 96, pp10-19) for a discussion.

Structure



Example

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. It is named after the singleton set, which is defined to be a set containing one element. The office of the President of the United States is a Singleton. The United States Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession. As a result, there can be at most one active president at any given time. Regardless of the personal identity of the active president, the title, "The President of the United States" is a global point of access that identifies the person in the office. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Opinions

"Singleton is not going to solve global data problems. When I think of Singleton, I think of a pattern that allows me to ensure a class has only one instance. As far as global data, the only benefit I see is reducing the namespace." [Christopher Parrinello]

"This is not really the point, but I always teach Composite, Strategy, Template Method, and Factory Method before I teach Singleton. They are much more common, and most people are probably already using the last two. The advantage of Singleton over global variables is that you are absolutely sure of the number of instances when you use Singleton, and, you can change your mind and manage any number of instances." [Ralph Johnson]

Our group had a bad habit of using global data, so I did a study group on Singleton. The next thing I know Singletons appeared everywhere and none of the problems related to global data went away. The answer to the global data question is not, "Make it a Singleton." The answer is, "Why in the hell are you using global data?" Changing the name doesn't change the problem. In fact, it may make it worse because it gives you the opportunity to say, "Well I'm not doing that, I'm doing this" - even though this and that are the same thing. [Frieder Knauss]

Rules of thumb

Abstract Factory, Builder, and Prototype can use Singleton in their implementation. [GOF, p134]

Facade objects are often Singletons because only one Facade object is required. [GOF, p193]

State objects are often Singletons. [GOF, p313]

<u>Program Examples</u>

Example 1:

```
// Purpose.  Singleton design pattern

// 1. Define a private static attribute in the "single instance" class
// 2. Define a public static accessor function in the class
// 3. Do "lazy initialization" (creation on demand) in the accessor function
// 4. Define all constructors to be protected or private
// 5. Clients may only use the accessor function to manipulate the Singleton
// 6. Inheritance can be supported, but static functions may not be overridden.
//    The base class must be declared a friend of the derived class (in order
//    to access the protected constructor).

#include <iostream>
#include <string>
#include <stdlib.h>
using namespace std;

class Number
{
public:
   static  Number* instance(); // 2. Define a public static accessor func
```

```
  static  void setType( string t ) { type = t;  delete inst;  inst = 0;}
  virtual void setValue( int in )  { value = in; }
  virtual int  getValue()         { return value; }
protected:
  int value;
  Number() { cout << ":ctor: "; }  // 4. Define all ctors to be protected
private:
  static string  type;
  static Number* inst;       // 1. Define a private static attribute
};

string  Number::type = "decimal";
Number* Number::inst = 0;

class Octal : public Number
{    // 6. Inheritance can be supported
public:
  friend class Number;
  void setValue( int in )
  {
    char buf[10];
    sprintf( buf, "%o", in );
    sscanf( buf, "%d", &value );
  }
protected:
  Octal() { }
};

Number* Number::instance()
{
  if ( ! inst)
    // 3. Do "lazy initialization" in the accessor function
    if (type == "octal")
           inst = new Octal();
    else
          inst = new Number();
  return inst;
}

void main( void )
{
  // Number  myInstance; --- error: cannot access protected constructor
  // 5. Clients may only use the accessor function to manipulate the Singleton
  Number::instance()->setValue( 42 );
  cout << "value is " << Number::instance()->getValue() << endl;
  Number::setType( "octal" );
  Number::instance()->setValue( 64 );
  cout << "value is " << Number::instance()->getValue() << endl;
}

// :ctor: value is 42
// :ctor: value is 100
```

PROGRAM 2:

```
// Purpose.  Singleton Discussion.  On the left, a global
// object is architected to require lazy initialization (not inited until it is needed).  //This
requires all users of the object to test and potentially allocate the //pointer.  Singleton
suggests making the class itself responsible for creating,
// maintaining, and providing global access to its own single instance.

#include <iostream.h>

class GlobalClass
{
public:
        GlobalClass( int v=0 )
         {
        value_ = v;
        }
        int  getValue()
        {
        return value_;
        }
        void setValue( int v )
        {
        value_ = v;
        }
private:
        int  value_;
};

// Initializing a global ptr to class GlobalClass
GlobalClass*   globalObj = 0;

void foo( void )
{
if ( ! globalObj )
globalObj = new GlobalClass;
globalObj->setValue( 1 );
cout << "foo: globalObj is " <<
globalObj->getValue() << endl;
}

void bar( void )
{
if ( ! globalObj )
globalObj = new GlobalClass;
globalObj->setValue( 2 );
cout << "bar: globalObj is " <<
globalObj->getValue() << endl;
}

void main( void )
{
if ( ! globalObj )
globalObj = new GlobalClass;
cout << "main: globalObj is " <<
globalObj->getValue() << endl;
foo();
bar();
}

// main: globalObj is 0
// foo: globalObj is 1
// bar: globalObj is 2
```

PROGRAM 3:

```
// Purpose.  Singleton destroyer Discussion.  Vlissides describes
// that Singletons can be cleaned-up by "wrapping" the ptr in a stack-
// based static member of another class whose sole responsibility is
// to have its destructor delete the Singleton's ptr.  The Singleton
// destroyer is automatically created before main() is run, and initially contains // a null
ptr. The first time the inst() method is called, the destroyer is
// meaningfully initialized.

#include <iostream.h>

class GlobalClass
{
public:
int  getValue()
 {
        return value_;
}



void setValue( int v )
 {
        value_ = v;
}
static GlobalClass* inst()
 {
        if ( ! globalObj_ )
        globalObj_ = new GlobalClass;
        return globalObj_;
}
protected:
GlobalClass( int v=0 )
{
        value_ = v;
}
~GlobalClass() { }
private:
        int    value_;
        static GlobalClass* globalObj_;
};

// Allocating and initializing
// GlobalClass's static data member
// (the ptr, not a GlobalClass inst)
```

```
GlobalClass*
GlobalClass::globalObj_ = 0;

void foo( void )
{
GlobalClass::inst()->setValue( 1 );
cout << "foo: globalObj is " << GlobalClass::inst()->getValue()<< endl;
}
void bar( void )
{
GlobalClass::inst()->setValue( 2 );
cout << "bar: globalObj is " <<GlobalClass::inst()->getValue()<< endl;
}

void main( void )
{
cout << "main: globalObj is " <<GlobalClass::inst()->getValue()<< endl;
foo();
bar();
}

// main: globalObj is 0
// foo: globalObj is 1
// bar: globalObj is 2
```

PROGRAM 4:

```
// New design.  "globalObj" is now a private static data member of its
// own class.  Global access is provided by the public static member
// function inst().  And the lazy initialization code is encapsulated in the inst()
// function. GlobalClass's ctor and dtor have been made protected so that
//clients cannot create more inst's or destroy the Singleton inst.

class GlobalClass
{
public:
int  getValue()
 {
        return value_;
}
void setValue( int v )
{
        value_ = v;
}
static GlobalClass* inst()
{
        if ( ! globalObj_ )
        globalObj_ = new GlobalClass;
        return globalObj_;
 }
protected:
GlobalClass( int v=0 )
{
        value_ = v;
}
~GlobalClass() { }
private:
        int    value_;
        static GlobalClass* globalObj_;
};

// Allocating and initializing GlobalClass's static data member
// (the ptr, not a GlobalClass inst)
GlobalClass*
GlobalClass::globalObj_ = 0;

void foo( void )
{
GlobalClass::inst()->setValue( 1 );
cout << "foo: globalObj is " <<GlobalClass::inst()->getValue()<< endl;
}

void bar( void )
{
GlobalClass::inst()->setValue( 2 );
cout << "bar: globalObj is " <<GlobalClass::inst()->getValue()<< endl;
}

void main( void )
{
cout << "main: globalObj is " <<GlobalClass::inst()->getValue()<< endl;
foo();
bar();
}

// main: globalObj is 0
// foo: globalObj is 1
// bar: globalObj is 2
```

PROGRAM 5:

```
class GlobalClass;

class SingDest
{
public:
SingDest( GlobalClass* s=0 )
{
        sing_ = s;
}
~SingDest();
void setSing( GlobalClass* s )
{
        sing_ = s;
}
private:
        GlobalClass*  sing_;
};

class GlobalClass
{
public:
        friend class SingDest;
        int  getValue() { return value_; }
        void setValue( int v )
        {
        value_ = v;
        }
static GlobalClass* inst()
{
if ( ! globalObj_ )
        {
        globalObj_ = new GlobalClass;
        dest_.setSing( globalObj_ );
        }
return globalObj_;
}
private:

GlobalClass( int v=0 )
{
cout << ":ctor: ";
value_ = v;
}

~GlobalClass()
{
cout << ":dtor:" << endl;
}

int    value_;
static GlobalClass* globalObj_;
static SingDest dest_;
};

GlobalClass* GlobalClass::globalObj_ = 0;
SingDest GlobalClass::dest_;
SingDest::~SingDest() { delete sing_; }

void foo( void )
{
GlobalClass::inst()->setValue( 1 );
cout << "foo: globalObj is " <<GlobalClass::inst()->getValue()<< endl;
}

void bar( void )
{
GlobalClass::inst()->setValue( 2 );
cout << "bar: globalObj is " <<GlobalClass::inst()->getValue()<< endl;
}
```

```
void main( void )
{
cout << "main: globalObj is " <<GlobalClass::inst()->getValue()<< endl;
foo();
bar();
cout << "main: end" << endl;
}

// main: globalObj is :ctor: 0
// foo: globalObj is 1
// bar: globalObj is 2
// main: end
// :dtor:
```

PROGRAM 6:

```
// Purpose.  Singleton design pattern lab.
//
// Problem.  The application would like a single instance of globalObject
// to exist, and chooses to implement it as a global.  Globals should
// always be discouraged.  Additionally, any code that references the
// global object, has to first check if the pointer has been initialized,
// and initialize it if it has not.
//
// Assignment.
// o Replace the global variable globalObject with a private static data member.
// o Provide the pattern-specified accessor function.
// o Provide for initialization and init testing in the GlobalObject class.
// o All client code should now use the Singleton accessor function instead of
//   referencing the globalObject variable.
// o Remove any client code dealing with globalObject initialization.
// o Guarantee that the GlobalObject class cannot be instantiated.

#include <iostream.h>

class GlobalObject
{
public:
   GlobalObject() : value_( 0 ) { }
   int  getValue()             { return value_; }
   void setValue( int val )    { value_ = val; }
private:
   int  value_;
};

GlobalObject*  globalObject;

void foo( void )
{
   if ( ! globalObject )
      globalObject = new GlobalObject;
   cout << "foo: globalObject's value is " << globalObject->getValue() << endl;
}

void bar( void )
{
   if ( ! globalObject )
      globalObject = new GlobalObject;
   globalObject->setValue( 42 );
   cout << "bar: globalObject's value is " << globalObject->getValue() << endl;
}

void main( void )
{
   foo();
   bar();
   if ( ! globalObject )
      globalObject = new GlobalObject;
   cout << "main: globalObject's value is " << globalObject->getValue() << endl;
}
```

```
// foo: globalObject's value is 0
// bar: globalObject's value is 42
// main: globalObject's value is 42
```

Adapter

Intent

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Problem

An "off the shelf" component offers compelling functionality that you would like reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.

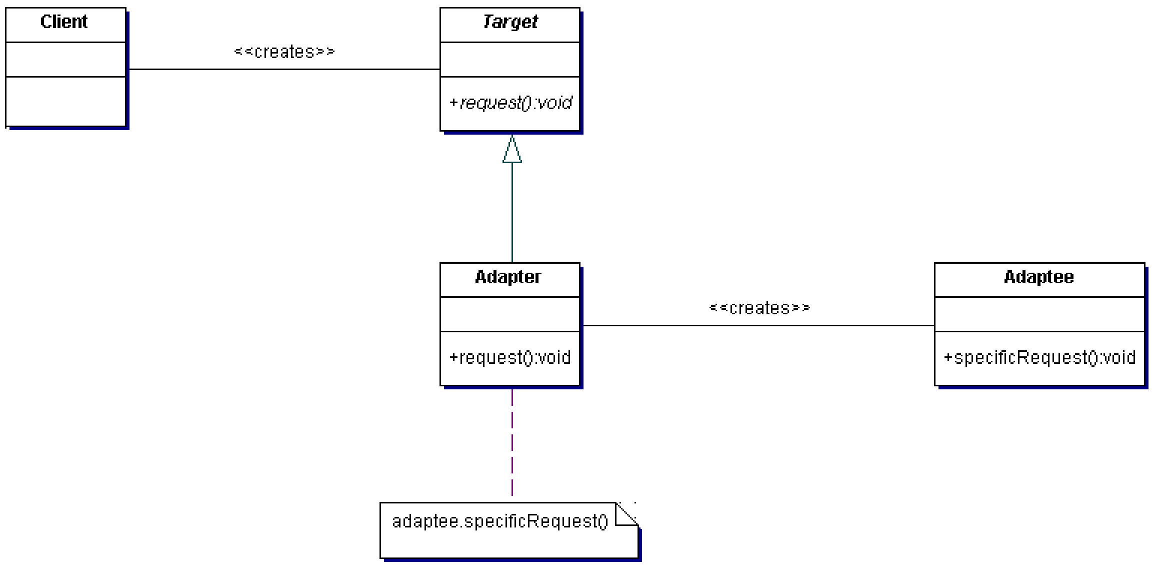
Discussion

Create an intermediary abstraction that translates, or maps, the old component to the new system. Clients call methods on the Adapter object which redirects them into calls to the legacy component. This strategy can be implemented either with inheritance or with aggregation.

Adapter functions as a wrapper or modifier of an existing class. It provides a different or translated view of that class. It effectively offers an impedence-matching facility.

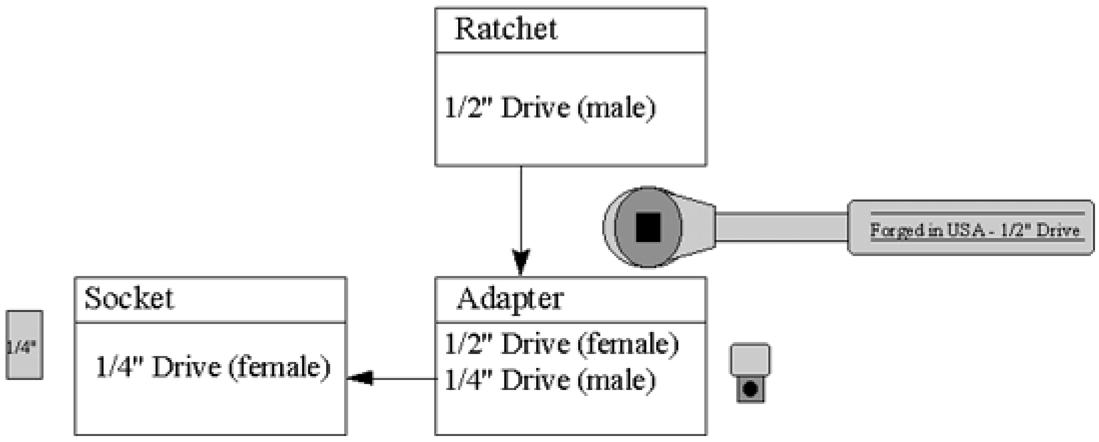
Structure

It is using aggregation is its delegation mechanism. The inheritance shown is only for interface - not for implementation.



Example

The Adapter pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients. Socket wrenches provide an example of the Adapter. A socket attaches to a ratchet, provided that the size of the drive is the same. Typical drive sizes in the United States are 1/2" and 1/4". Obviously, a 1/2" drive ratchet will not fit into a 1/4" drive socket unless an adapter is used. A 1/2" to 1/4" adapter has a 1/2" female connection to fit on the 1/2" drive ratchet, and a 1/4" male connection to fit in the 1/4" drive socket. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

Adapter makes things work after they're designed; Bridge makes them work before they are. [GOF, p219]

Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together. [GOF, p161]

Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface. [GOF. p216]

Adapter is meant to change the interface of an existing object. Decorator enhances another object without changing its interface. Decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters. [GOF, 149]

Facade defines a new interface, whereas Adapter reuses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one. [GOF, pp219]

Program examples

Example 1:

```
// Purpose.  Adapter design pattern lab
//
// Problem.  Simulated here is a "stack machine", or a non-OO
// implementation of a Stack class.  We would like to reuse this "legacy"
// asset as the basis for a new Queue class, but there is an "impedance
// mismatch" between the old and the new.
//
// Assignment.
//  The class Queue should contain an "instance" of component Stack.
//  Queue needs a ctor, dtor, enque, deque, and isEmpty methods.  Each of
//   these methods will provide an adaptation from the Queue interface to the
//   Stack implementation.
//  If you map Queue's enque() to Stack's push(), then the pseudocode below
//  will give you insight into implementing Queue's deque() method.

#include <iostream.h>

struct StackStruct
{
   int*  array;
   int   sp;
   int   size;
};
typedef StackStruct Stack;

static void initialize( Stack* s, int size )
{
   s->array = new int[size];
   s->size = size;
   s->sp = 0;
}
static void cleanUp( Stack* s )
{
  delete s->array;
}
static int isEmpty( Stack* s )
{
   return s->sp == 0 ? 1 : 0;
}
```

```
static int isFull( Stack* s )
{
  return s->sp == s->size ? 1 : 0;
}
static void push( Stack* s, int item )
{
  if ( ! isFull(s)) s->array[s->sp++] = item;
 }
static int pop( Stack* s )
 {
  if (isEmpty(s))
          return 0;
  else
          return s->array[--s->sp];
}

class Queue
{
  deque pseudocode:
    initialize a local temporary instance of Stack
    loop
      pop() the permanent stack and push() the temporary stack
    pop() the temporary stack and remember this value
    loop
      pop() the temporary stack and push() the permanent stack
    cleanUp the temporary stack
    return the remembered value
  int isFull()
           {
                    return ::isFull( &_stack );
          }
};

void main( void )
{
  Queue  queue(15);

  for (int i=0; i < 25; i++) queue.enque( i );
  while ( ! queue.isEmpty())
    cout << queue.deque() << " ";
  cout << endl;
}

// 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

Example 2:

```
// Purpose.  Adapter design pattern demo
//
// Discussion.  LegacyRectangle's interface is not compatible with the
// system that would like to reuse it.  An abstract base class is created
// that specifies the desired interface.  An "adapter" class is defined
// that publicly inherits the interface of the abstract class, and
// privately inherits the implementation of the legacy component.  This
// adapter class "maps" or "impedance matches" the new interface to the
// old implementation.

#include <iostream.h>

typedef int Coordinate;
typedef int Dimension;

/////////////////////////// Desired interface ///////////////////////////
class Rectangle
{
public:
  virtual void draw() = 0;
};

/////////////////////////// Legacy component ///////////////////////////
class LegacyRectangle
{
public:
  LegacyRectangle( Coordinate x1, Coordinate y1,  Coordinate x2, Coordinate y2 )
{
    x1_ = x1;  y1_ = y1;  x2_ = x2;  y2_ = y2;
    cout << "LegacyRectangle:  create.  (" << x1_ << "," << y1_<< ") =>
                                        (" << x2_ << "," << y2_ << ")" << endl;
}

void oldDraw()
{
    cout << "LegacyRectangle:  oldDraw.  (" << x1_ << "," << y1_
      << ") => (" << x2_ << "," << y2_ << ")" << endl;
}
private:
  Coordinate x1_;
  Coordinate y1_;
  Coordinate x2_;
  Coordinate y2_;
};

/////////////////////////// Adapter wrapper ///////////////////////////
class RectangleAdapter : public Rectangle, private LegacyRectangle
{
public:
  RectangleAdapter( Coordinate x, Coordinate y, Dimension w, Dimension h )
      : LegacyRectangle( x, y, x+w, y+h )
{
      cout << "RectangleAdapter: create.  (" << x << "," << y << "), width = " << w << ",
                                                            height = " << h << endl;
}
   virtual void draw()
{
      cout << "RectangleAdapter: draw." << endl;
      oldDraw(); }
};

void main()
{
   Rectangle*  r = new RectangleAdapter( 120, 200, 60, 40 );
   r->draw();
}

// LegacyRectangle:  create.  (120,200) => (180,240)
// RectangleAdapter: create.  (120,200), width = 60, height = 40
// RectangleAdapter: draw.
// LegacyRectangle:  oldDraw.  (120,200) => (180,240)
```

Example 3:

```
// Purpose.  Adapter design pattern (External Polymorphism demo)

// 1. Specify the new desired interface
// 2. Design a "wrapper" class that can "impedance match" the old to the new
// 3. The client uses (is coupled to) the new interface
// 4. The adapter/wrapper "maps" to the legacy implementation

#include <iostream.h>

class ExecuteInterface
{
public:                  // 1. Specify the new i/f
   virtual ~ExecuteInterface() { }
   virtual void execute() = 0;
};

template <class TYPE>                                    // 2. Design a "wrapper" or
class ExecuteAdapter : public ExecuteInterface //    "adapter" class
{  public:
   ExecuteAdapter( TYPE* o, void (TYPE::*m)() ) { object = o;  method =m; }
   ~ExecuteAdapter()                            { delete object; }
   // 4. The adapter/wrapper "maps" the new to the legacy implementation
   void execute()             /* the new */
   {
             (object->*method)();
   }
private:
   TYPE* object;                                // ptr-to-object attribute
   void (TYPE::*method)();    /* the old */       // ptr-to-member-function
};                                             //   attribute
```

```
// The old: three totally incompatible classes    // no common base class,
class Fea
{
 public:                              // no hope of polymorphism
   ~Fea()       { cout << "Fea::dtor" << endl; }
   void doThis() { cout << "Fea::doThis()" << endl; }
};

class Feye
{
public:
   ~Feye()      { cout << "Feye::dtor" << endl; }
   void doThat() { cout << "Feye::doThat()" << endl; }
};

class Pheau
{
public:
   ~Pheau()         { cout << "Pheau::dtor" << endl; }
   void doTheOther() { cout << "Pheau::doTheOther()" << endl; }
};

/* the new is returned */ ExecuteInterface** initialize()
{
   ExecuteInterface** array = new ExecuteInterface*[3]; /* the old is below */
   array[0] = new ExecuteAdapter<Fea>(   new Fea(),     &Fea::doThis       );
   array[1] = new ExecuteAdapter<Feye>(  new Feye(),    &Feye::doThat      );
   array[2] = new ExecuteAdapter<Pheau>( new Pheau(),   &Pheau::doTheOther );
   return array;
}

void main( void )
{
   ExecuteInterface** objects = initialize();

   for (int i=0; i < 3; i++) objects[i]->execute();  // 3. Client uses the new
                                              //    (polymporphism)
   for (i=0; i < 3; i++) delete objects[i];
   delete objects;
}

// Fea::doThis()
// Feye::doThat()
// Pheau::doTheOther()
// Fea::dtor
// Feye::dtor
// Pheau::dtor
```

Example 4:

```
// Purpose.  Adapter
//
// Discussion.  The Adapter pattern discusses how to "wrap" the old in-
// terface of a legacy component, so that it can continue to contribute
// in a new system.  It is about "impedance matching" an old dog with
// new tricks (to mix metaphors).  On the left, WimpyTime "hasa" in-
// stance of the legacy component, and delegates the "heavy lifting"
// to it.  On the right, private derivation is used to accomplish the same result.

class ManlyTime
{
public:
char* getTime()
        {
        static char buf[30];
        time_t  lt;
        tm*     ltStruct;
        time( &lt );
        ltStruct = localtime(&lt);
        strftime( buf, 30, "%H%M",ltStruct );
        return buf;
        }
};

class WimpyTime
{
public:
char* getTime()
        {
        static char buf[30];
        char *ptr, mi[3], am[3];
        int  hr;
        ptr = imp_.getTime();
        cout << "old interface time is "<< ptr << endl;
        strcpy( mi, &(ptr[2]) );
        ptr[2] = '\0';
        sscanf( ptr, "%d", &hr );
        strcpy( am, "AM" );
        if (hr > 12)
         {
                hr -= 12;
                strcpy( am, "PM" ); }
                sprintf( buf, "%d:%s %s", hr, mi, am );
                return buf;
        }
private:
        ManlyTime  imp_;
};
```

```
void main( void )
{
WimpyTime  newT;
char*      ptr;
ptr = newT.getTime();
cout << "new interface time is "
<< ptr << endl;
}

// old interface time is 1709
// new interface time is 5:09 PM
```

Example 5:

```
#include <iostream.h>
#include <stdio.h>
#include <time.h>
class ManlyTime
{
public:
char* getTime()
        {
        static char buf[30];
        time_t  lt;
        tm*     ltStruct;
        time( &lt );
        ltStruct = localtime(&lt);
        strftime( buf, 30, "%H%M",ltStruct );
        return buf;
        }
};

class WimpyTime :private ManlyTime
{
public:
char* getTime()
        {
        static char buf[30];
        char *ptr, mi[3], am[3];
        int  hr;
        ptr = ManlyTime::getTime();
        cout << "old interface time is "<< ptr << endl;
        strcpy( mi, &(ptr[2]) );
        ptr[2] = '\0';
        sscanf( ptr, "%d", &hr );
        strcpy( am, "AM" );
        if (hr > 12)
         {
                hr -= 12;
                strcpy( am, "PM" ); }
                sprintf( buf, "%d:%s %s",hr, mi, am );
                return buf;
        }
};
```

```
void main( void )
{
WimpyTime  newT;
char*      ptr;
ptr = newT.getTime();
cout << "new interface time is "
<< ptr << endl;
}

// old interface time is 1721
// new interface time is 5:21 PM
```

Bridge

Intent

Decouple an abstraction from its implementation so that the two can vary independently.

Problem

"Hardening of the software arteries" has occurred by using subclassing of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be independently extended or composed.

Discussion

Decompose the component's interface and implementation into orthogonal class hierarchies. The interface class contains a pointer to the abstract implementation class. This pointer is initialized with an instance of a concrete implementation class, but all subsequent interaction from the interface class to the implementation class is limited to the abstraction maintained in the implementation base class. The client interacts with the interface class, and it in turn "delegates" all requests to the implementation class.

The interface object is the "handle" known and used by the client; while the implementation object, or "body", is safely encapsulated to ensure that it may continue to evolve, or be entirely replaced (or shared at run-time.

Use the Bridge pattern when:

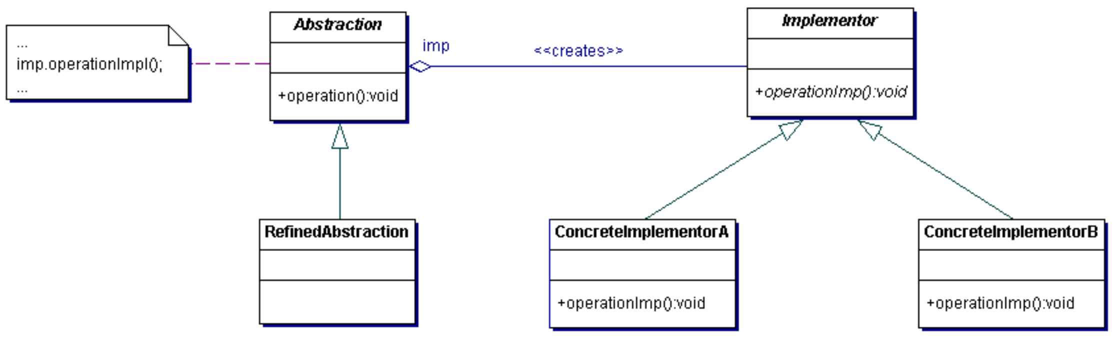
- you want run-time binding of the implementation,
- you have a proliferation of classes resulting from a coupled interface and numerous implementations,
- you want to share an implementation among multiple objects,
- you need to map orthogonal class hierarchies.

Consequences include:

- decoupling the object's interface,
- improved extensibility (you can extend (i.e. subclass) the abstraction and implementation hierarchies independently),
- hiding details from clients.

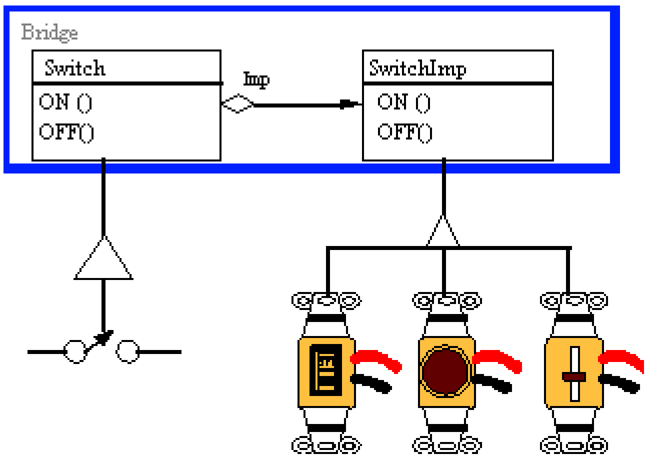
Bridge is a synonym for the "handle/body" idiom [Coplien, *C++ Report*, May 95, p58]. This is a design mechanism that encapsulates an implementation class inside of an interface class. The former is the body, and the latter is the handle. The handle is viewed by the user as the actual class, but the work is done in the body. "The handle/body class idiom may be used to decompose a complex abstraction into smaller, more manageable classes. The idiom may reflect the sharing of a single resource by multiple classes that control access to it (e.g. reference counting)." [Coplien, *Advanced C++*, p62]

Structure



Example

The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

Adapter makes things work after they're designed; Bridge makes them work before they are. [GOF, p219]

Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together. [GOF, 216]

State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom [Coplien, *Advanced C++*, p58]. They differ in intent - that is, they solve different problems.

The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one). The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently. [Coplien, *C++ Report*, May 95, p58]

If interface classes delegate the creation of their implementation classes (instead of creating/coupling themselves directly), then the design usually uses the Abstract Factory pattern to create the implementation objects. [Grand, *Patterns in Java*, p196]

Program Examples

Example 1:

```
// Purpose.  Bridge design pattern lab
//
// Problem.  Our Stack class needs to allow the client to specify at
// run-time what implementation strategy to employ (i.e. list vs array).
//
// Assignment.
// o Rename the current Stack class to ArrayImp
// o Create a base class called StackImp
// o StackImp is a "pure abstract base class".  It should declare 4 pure
//   virtual functions: push(), pop(), isEmpty(), and isFull()
// o Derive classes ArrayImp and ListImp from class StackImp
// o Create an interface (or wrapper, or handle) class called Stack
// o The class Stack should contain a pointer to the abstract base class
//   StackImp.
// o To initialize the StackImp pointer, Stack needs a constructor that
//   accepts one argument of type ImplementationType.  In the body, you
//   create an instance of the derived class requested.  If the client does
//   not supply an argument for the constructor, provide a default argument
//   of ArrayImplementation.
// o Don't forget to write a destructor to clean-up the StackImp pointer
// o The class Stack should have a method for anything a client might want to
//   do to a Stack object.  Each method simply forwards the client request to
//   the appropriate method in its StackImp object.
// o Un-comment the code in main() and test your results

#include <iostream.h>
enum ImplementationType { ArrayImplementation, ListImplementation };

class Stack
{
public:
   Stack( int = 10 );
   ~Stack();
   void push( int val );
   int  pop();
   int  isEmpty();
   int  isFull();
private:
   int* _array;
   int  _sp;
   int  _size;
};


class Node
{
public:
   Node( int val, Node* next );
   int getValue();
   Node* getNext();
private:
   int    _value;
   Node*  _next;
};

class ListImp
{
public:
    ListImp();
    ~ListImp();
    void push( int val );
    int  pop();
    int  isEmpty();
    int  isFull();
private:
    Node* _head;
};
```

```
void main ( void )
{
   Stack stack1;
   // Stack stack2( ListImplementation );

   for (int i=1; i < 16; i++) {
      stack1.push( i );
      // stack2.push( i );
}

   cout << "Array stack: ";
   for (i=1; i < 18 ; i++)
      cout << stack1.pop() << "  ";
   cout << endl;

   // cout << "List stack:  ";
   // for (i=1; i < 18 ; i++)
   //    cout << stack2.pop() << "  ";
   // cout << endl;
}

// Array stack: 10  9  8  7  6  5  4  3  2  1  0  0  0  0  0  0  0
// List stack:  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0  0

//************* DEFINITIONS************************//
Stack::Stack( int size )
{
   _array = new int[size];
   _size = size;
   _sp = 0; }
Stack::~Stack() { delete _array; }
void Stack::push( int val )
{
   if ( ! isFull())
      _array[_sp++] = val;
}
int Stack::pop()
{
   if (isEmpty())
         return 0;
   else
         return _array[--_sp];
}
int Stack::isEmpty() { return _sp == 0 ? 1 : 0; }
int Stack::isFull()  { return _sp == _size ? 1 : 0; }

Node::Node( int val, Node* next )
{
   _value = val;
   _next = next;
}
int   Node::getValue() { return _value; }
Node* Node::getNext()  { return _next; }

ListImp::ListImp() { _head = NULL; }
```

```
ListImp::~ListImp()
{
   Node  *current, *previous;
   current = _head;
   while (current)
          {
               previous = current;
               current = current->getNext();
               delete previous;
          }
}
void ListImp::push( int val )
{
   Node*  temp = new Node( val, _head );
   _head = temp;
}

int ListImp::pop()
{
   if (isEmpty()) return 0;
   Node*  temp = _head;
   int    val  = _head->getValue();
   _head = _head->getNext();
   delete temp;
   return val;
}
int ListImp::isEmpty() { return _head ? 0 : 1; }
int ListImp::isFull()  { return 0; }
```

Example 2:

```
// Purpose.  Bridge design pattern demo
// Discussion.  The motivation is to decouple the Time interface from the
// Time implementation, while still allowing the abstraction and the
// realization to each be modelled with their own inheritance hierarchy.
// The implementation classes below are straight-forward.  The interface
// classes are a little more subtle.  Routinely, a Bridge pattern
// interface hierarchy "hasa" implementation class.  Here the interface
// base class "hasa" a pointer to the implementation base class, and each
// class in the interface hierarchy is responsible for populating the base
// class pointer with the correct concrete implementation class.  Then all
// requests from the client are simply delegated by the interface class to
// the encapsulated implementation class.

#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class TimeImp
{
public:
   TimeImp( int hr, int min )
{
     hr_ = hr;  min_ = min;
}
```

```
virtual void tell()
{
    cout << "time is " << setw(2) << setfill(48) << hr_ << min_ << endl;
}
protected:
  int hr_, min_;
};

class CivilianTimeImp : public TimeImp
{
public:
  CivilianTimeImp( int hr, int min, int pm ) : TimeImp( hr, min )
{
    if (pm)
      strcpy( whichM_, " PM" );
    else
      strcpy( whichM_, " AM" ); }
  /* virtual */ void tell()
{
    cout << "time is " << hr_ << ":" << min_ << whichM_ << endl;
}
protected:
  char  whichM_[4];
};

class ZuluTimeImp : public TimeImp
{
public:
  ZuluTimeImp( int hr, int min, int zone ) : TimeImp( hr, min )
{
    if (zone == 5)
      strcpy( zone_, " Eastern Standard Time" );
    else if (zone == 6)
      strcpy( zone_, " Central Standard Time" );
}
 /* virtual */ void tell()
{
    cout << "time is " << setw(2) << setfill(48) << hr_ << min_ << zone_ << endl;
}
protected:
  char  zone_[30];
};

class Time
{
public:
  Time() { }
  Time( int hr, int min )
        {
            imp_ = new TimeImp( hr, min );
        }
  virtual void tell()
        {
            imp_->tell();
        }
```

```
protected:
  TimeImp*  imp_;
};

class CivilianTime : public Time
{
public:
  CivilianTime( int hr, int min, int pm )
        {
            imp_ = new CivilianTimeImp( hr, min, pm );
        }
};

class ZuluTime : public Time
{
public:
  ZuluTime( int hr, int min, int zone )
        {
            imp_ = new ZuluTimeImp( hr, min, zone );
        }
};

void main()
{
  Time*  times[3];
  times[0] = new Time( 14, 30 );
  times[1] = new CivilianTime( 2, 30, 1 );
  times[2] = new ZuluTime( 14, 30, 6 );
  for (int i=0; i < 3; i++)
    times[i]->tell();
}

// time is 1430
// time is 2:30 PM
// time is 1430 Central Standard Time
```

Example 3:

```
// Purpose.  Bridge
//
// Discussion.  Even though Date has a clean interface and a well encap-
// sulated implementation, the client still has to recompile if the class
// architect changes his/her mind. Instead, create a wrapper (or inter-
// face) class that contains and delegates to a body (or implementation)
// class.  Client can now specify at run-time exactly what s/he wants.

#include <iostream.h>
#include <stdio.h>

class Date
{
public:
Date( int y, int m, int d );
void output();
private:
        #ifdef OK
                int  year_, month_, day_;
        #endif

        #ifdef AA
                int        toJulian(int,int,int);
                char*      fromJulian(void);
                int        julian_;
                int        year_;
                static int dayTable_[2][13];
        #endif

};

#ifdef OK
void Date::output()
{
char buf[20];
int year = year_ - (year_/100*100);
sprintf( buf, "%02d%02d%02d",year, month_, day_ );
cout << buf << "  ";
}
#endif

#ifdef AA
void Date::output()
{
cout << fromJulian() << "  ";
}
#endif

#include "bridge1.inc"

void main( void )
{
Date  d1( 1996, 2, 29 );
Date  d2( 1996, 2, 30 );
d1.output();
d2.output();
cout << endl;
}

// 960229  960230
// 960229  960301
```

Example 4:

```
class DateImp;

class Date
{
public:
        Date( int y, int m, int d );
        ~Date();
        void output();
        static void setImp( char* t )
        {
                strcpy( impType_, t );
        }
private:
        DateImp*    rep_;
        static char impType_[10];
};
char Date::impType_[] = "Ok";

class DateImp
{
 public:
        virtual void output() = 0;
};

class DateOk : public DateImp
{
public:
        DateOk( int y, int m, int d );
        void output();
private:
        int  year_, month_, day_;
};

class DateAA : public DateImp
{
public:
        DateAA( int y, int m, int d );
        void output();
private:
        int     toJulian(int,int,int);
        char*     fromJulian(void);
        int     julian_;
        int     year_;
        static int dayTable_[2][13];
};

Date::Date( int y, int m, int d )
{
        if ( ! strcmp( impType_, "Ok" ))
                rep_ = new DateOk( y, m, d );
        else
                rep_ = new DateAA( y, m, d );
}
```

```
Date::~Date()       { delete rep_; }
void Date::output() { rep_->output(); }

#include "bridge2.inc"

void main( void )
{
Date  d1( 1996, 2, 29 );
Date  d2( 1996, 2, 30 );
Date::setImp( "AA" );
Date  d3( 1996, 2, 29 );
Date  d4( 1996, 2, 30 );
d1.output();  d2.output();
cout << endl;
d3.output();  d4.output();
cout << endl;
}

// 960229  960230
// 960229  960301
```

Bridge pattern = Insulation

encapsulation

implementation details (type, data, or function) are not accessible programmatically through the interface of the component - a logical property of design

insulation

implementation details (type, data, or function) can be altered without forcing clients of the component to recompile - a physical property of design

Carroll and Ellis - inheritance hierarchy styles

The design of a reusable library can be based on one of several inheritance hierarchy styles or on a combination of styles. These include:

- Direct hierarchy

  Design as usual - map domain types directly to C++ classes. Intermixes interface and implementation. Minimizes the number of classes. Implementation changes generally cause link incompatiblilities.

- Interfaced hierarchy

  Mirror every direct hierarchy class with a pure abstract class (interface only). This doubles the number of classes but improves extensibility and link compatibility.

- Interfaced + Factory hierarchy

  Add an "object factory" to the interfaced hierarchy. Each factory encapsulates all client creation services (clients never "new" their own instances). Clients can now write programs that will need no recompiling whatsoever to upgrade to a new release of the library.

- Handle hierarchy

  Wrap every direct hierarchy class with a "handle" class. The original class becomes a hidden "body" or "representation" class. Each "handle" class is a thin veneer that simply delegates all user requests to its contained "body" class. Any change to the implementation of a "body" class will be link compatible for clients of its encapsulating "handle" class. Not as extensible as Direct and Interfaced hierarchies.

- Interfaced Handle hierarchy

  Add a third set of interface classes (body classes are the first set, and handle classes are the second set) to reinstate some amount of extensibility.

Although a direct inheritance hierarchy is the easiest style to implement and understand as well as the most efficient, interfaced hierarchies, object factories, and handle hierarchies facilitate link compatibility between releases of a library. Further, interfaced hierarchies increase a library's extensibility. The table summarizes the most important differences among the hierarchy styles. As always, no single design is best for all libraries. Library designers must decide which is the best choice for their library and their users.

Hierarchy style	**Complexity**	**Efficiency**	**Extensibility**	**Link compatibility**
Direct	simple	good	mediocre	minimal
Interfaced	complex	reduced	good	partial
Interfaced + Factory	complex	reduced	good	total
Handle	simple	reduced	poor	total
Interfaced Handle	complex	reduced	good	total

Lakos - insulation techniques

Handle

- use when concrete usage of the class predominates
- use when you are already creating instances of the class

Interfaced

- allows the interface to be subscribed to without picking up any implementation baggage
- use when abstract usage of the class predominates
- use when you are only referring to the class via the base class

Meyers - insulation techniques

Handle

- defer to contained pointer
- easy to switch between Handle and Body classes with *typedef*
- Handle operations can't be inlined
- must pay for heap allocation of Body instance
- difficult to extend with inheritance

Interfaced

- defer to v-table pointer
- good with inheritance
- Interface operations can't be inlined
- must pay for heap allocation of derived instances
- no easy way to switch between Interface and derived classes

Composite

Intent

Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Problem

Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.

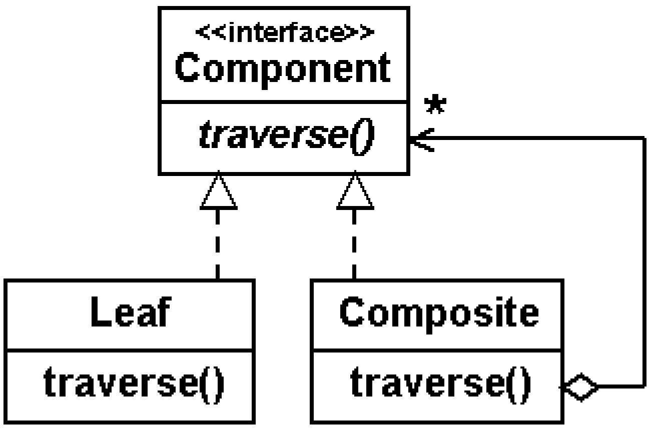
Discussion

Define an abstract base class (Component) that specifies the behavior that needs to be exercised uniformly across all primitive and composite objects. Subclass the Primitive and Composite classes off of the Component class. Each Composite object "couples" itself only to the abstract type Component as it manages its "children".

Use this pattern whenever you have "composites that contain components, each of which could be a composite".

Child management methods [e.g. addChild(), removeChild()] should normally be defined in the Composite class. Unfortunately, the desire to treat Primitives and Composites uniformly requires that these methods be moved to the abstract Component class. See the "Opinions" section below for a discussion of "safety" versus "transparency" issues.

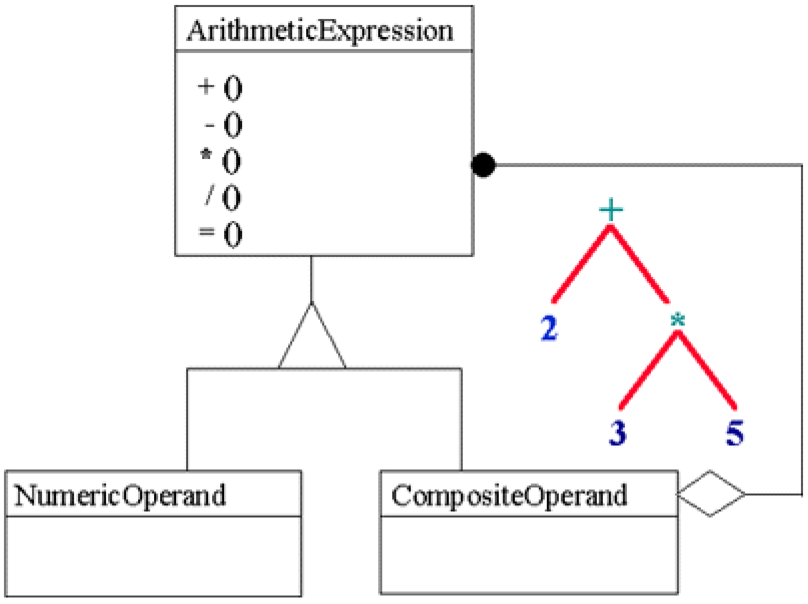
Structure



Note that it is missing the "recursive composition" relationship from *Composite* to *Component*.

Example

The Composite composes objects into tree structures and lets clients treat individual objects and compositions uniformly. Although the example is abstract, arithmetic expressions are Composites. An arithmetic expression consists of an operand, an operator (+ - * /), and another operand. The operand can be a number, or another arithmetic expresssion. Thus, 2 + 3 and (2 + 3) + (4 * 6) are both valid expressions. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects. [GOF, p219]

Composite can be traversed with Iterator. Visitor can apply an operation over a Composite. Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition. It could use Observer to tie one object structure to another and State to let a component change its behavior as its state changes. [GOF, pp173,349]

Composite can let you compose a Mediator out of smaller pieces through recursive composition. [Vlissides, Apr96, p18]

Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert. [GOF, p220]

Flyweight is often combined with Composite to implement shared leaf nodes. [GOF, p206]

Opinions

The whole point of the Composite pattern is that the Composite can be treated atomically, just like a leaf. If you want to provide an Iterator protocol, fine, but I think that is outside the pattern itself. At the heart of this pattern is the ability for a client to perform operations on an object without needing to know that there are many objects inside. [Don Roberts]

Being able to treat a heterogeneous collection of objects atomically (or transparently) requires that the "child management" interface be defined at the root of the Composite class hierarchy (the abstract Component class). However, this choice costs you safety, because clients may try to do meaningless things like add and remove objects from leaf objects. On the other hand, if you "design for safety", the child management interface is declared in the Composite class, and you lose transparency because leaves and Composites now have different interfaces. [Bill Burcham]

Smalltalk implementations of the Composite pattern usually do not have the interface for managing the components in the Component interface, but in the Composite interface. C++ implementations tend to put it in the Component interface. This is an extremely interesting fact, and one that I often ponder. I can offer theories to explain it, but nobody knows for sure why it is true. [Ralph Johnson]

My Component classes do not know that Composites exist. They provide no help for navigating Composites, nor any help for altering the contents of a Composite. This is because I would like the base class (and all its derivatives) to be reusable in contexts that do not require Composites. When given a base class pointer, if I absolutely need to know whether or not it is a Composite, I will use dynamic_cast to figure this out. In those cases where dynamic_cast is too expensive, I will use a Visitor. [Robert Martin]

Common complaint: "if I push the Composite interface down into the Composite class, how am I going to enumerate (i.e. traverse) a complex structure?" My answer is that when I have behaviors which apply to hierarchies like the one presented in the Composite pattern, I typically use Visitor, so enumeration isn't a problem - the Visitor knows in each case, exactly what kind of object it's dealing with. The Visitor doesn't need every object to provide an enumeration interface. [Bill Burcham]

Composite doesn't force you to treat all Components as Composites. It merely tells you to put all operations that you want to treat "uniformly" in the Component class. If add, remove, and similar operations cannot, or must not, be treated uniformly, then do not put them in the Component base class. Remember, by the way, that each pattern's structure diagram doesn't define the pattern; it merely depicts what in our experience is a common realization thereof. Just because Composite's structure diagram shows child management operations in the Component base class doesn't mean all implementations of the pattern must do the same. [John Vlissides]

Program Examples

Example 1:

```
// Purpose.  Composite

#include <string.h>
enum NodeType { FileT, DirT };
int  g_indent = 0;

class File
{
public:
File( char* n )
         {
                  type_ = FileT;
                 strcpy( name_, n );
        }
NodeType getType()
          {
                     return type_;
          }
void ls()
          {
          for (int i=0; i < g_indent; i++)
                    cout << ' ';
          cout << name_ << endl;
          }
private:
          NodeType  type_;
          char      name_[20];
};

class Dir
{
public:
Dir( char* n )
          {
                     type_ = DirT;
                    strcpy( name_, n );
                     total_ = 0;
          }
NodeType getType()
          {
                    return type_;
          }
void add( File* f )
          {
                    files_[total_++] = f;
          }
void ls()
          {
                    for (int i=0; i < g_indent; i++)
                              cout << ' ';
                    cout << name_ << ":" << endl;
                    g_indent += 3;
                    for (int i=0; i < total_; i++)
                              if (files_[i]->getType()== DirT)
                                        ((Dir*) files_[i])->ls();
                              else
                                        files_[i]->ls();
                    g_indent -= 3;
          }
private:
          NodeType  type_;
          char      name_[20];
          File*     files_[10];
          int       total_;
};
```

```
void main( void )
{
Dir   one("1"), two("2"), thr("3");
File  a("a"), b("b"), c("c"),
d("d"), e("e");
one.add( &a );
one.add( (File*) &two );
one.add( &b );
two.add( &c );
two.add( &d );
two.add( (File*) &thr );
thr.add( &e );
one.ls();
}

// 1:        //      d
//    a      //      3:
//    2:     //         e
//       c   //    b
```

Example 2:

```
// Strategy.  Use recursive composition to create a heterogeneous aggregate
// that can be treated homogeneously.
// Benefit.  No more type checking and type casting (coupling between Dir
// and File is gone, Dir is only coupled to abstract base class)

class AbsFile
{
public:
        virtual void ls() = 0;
protected:
        char       name_[20];
        static int  indent_;
};
int AbsFile::indent_ = 0;

class File: public AbsFile
{
public:
File( char* n )
        {
                strcpy( name_, n );
        }
void ls()
        {
                for (int i=0; i < indent_; i++)
                        cout << ' ';
                cout << name_ << endl;
        }
};
```

```
class Dir : public AbsFile
{
public:
Dir( char* n )
        {
                strcpy( name_, n ); total_ = 0;
        }
void add( AbsFile* f )
        {
                files_[total_++] = f;
        }
void ls()
        {
                for (int i=0; i < indent_; i++)
                        cout << ' ';
                cout << name_ << ":" << endl;
                indent_ += 3;
                for (int i=0; i < total_; i++)
                        files_[i]->ls();
                indent_ -= 3;
        }
private:
        AbsFile*  files_[10];
        int       total_;
};

void main( void )
{
Dir   one("1"), two("2"), thr("3");
File  a("a"), b("b"), c("c"),
d("d"), e("e");
one.add( &a );
one.add( &two );
one.add( &b );
two.add( &c );
two.add( &d );
two.add( &thr );
thr.add( &e );
one.ls();
}

// 1:            //         d
//    a          //         3:
//    2:         //            e
//       c       //      b
```

Example 3:

```
// Purpose.  Composite design pattern
// 1. Identify the scalar/primitive classes and vector/container classes
// 2. Create an "interface" (lowest common denominator) that can make all
//    concrete classes "interchangeable"
// 3. All concrete classes declare an "is a" relationship to the interface
// 4. All "container" classes couple themselves to the interface (recursive
//    composition, Composite "has a" set of children up the "is a" hierarchy)
// 5. "Container" classes use polymorphism as they delegate to their children

#include <iostream>
#include <vector>
using namespace std;

// 2. Create an "interface" (lowest common denominator)
class Component { public: virtual void traverse() = 0; };

class Leaf : public Component
{
      // 1. Scalar class   3. "isa" relationship
      int value;
public:
   Leaf( int val ) { value = val; }
   void traverse() { cout << value << ' '; }
};

class Composite : public Component
{
   // 1. Vector class   3. "isa" relationship
   vector<Component*> children;        // 4. "container" coupled to the interface
public:
   // 4. "container" class coupled to the interface
   void add( Component* ele ) { children.push_back( ele ); }
   void traverse()
         {
               for (int i=0; i < children.size(); i++)
             // 5. Use polymorphism to delegate to children
                  children[i]->traverse();
         }
};

void main( void )
{
   Composite containers[4];

   for (int i=0; i < 4; i++)
      for (int j=0; j < 3; j++)
         containers[i].add( new Leaf( i * 3 + j ) );

   for (i=1; i < 4; i++)
      containers[0].add( &(containers[i]) );

   for (i=0; i < 4; i++)
         {
               containers[i].traverse();
               cout << endl;
         }
}

// 0 1 2 3 4 5 6 7 8 9 10 11
// 3 4 5
// 6 7 8
// 9 10 11
```

Example 4:

```
// Purpose.  Composite design pattern - multiple container classes

#include <iostream>
#include <vector>
using namespace std;

class Component { public: virtual void traverse() = 0; };
class Primitive : public Component
{
   int value;
public:
   Primitive( int val ) { value = val; }
   void traverse()      { cout << value << "  "; }
};

class Composite : public Component
{
   vector<Component*> children;
   int                value;
public:
   Composite( int val )     { value = val; }
   void add( Component* c ) { children.push_back( c ); }
   void traverse()
         {
               cout << value << "  ";
               for (int i=0; i < children.size(); i++)
                           children[i]->traverse();
         }
 };

class Row : public Composite
{
 public:     // Two different kinds of "con-
   Row( int val ) : Composite( val ) { }   // tainer" classes.  Most of the
   void traverse()
         {                    // "meat" is in the Composite
               cout << "Row";                      // base class.
               Composite::traverse();
         }
 };

class Column : public Composite
{
 public:
   Column( int val ) : Composite( val ) { }
   void traverse()
         {
               cout << "Col";
               Composite::traverse();
         }
  };
```

```
void main( void )
{
    Row    first(1);                        // Row1
    Column second( 2 );                     //   |
    Column third( 3 );                      //   +-- Col2
    Row    fourth( 4 );                     //   |      |
    Row    fifth( 5 );                      //   |      +-- 7
    first.add( &second );                   //   +-- Col3
    first.add( &third  );                   //   |      |
    third.add( &fourth );                   //   |      +-- Row4
    third.add( &fifth  );                   //   |      |      |
    first.add(  &Primitive( 6 ) );          //   |      |      +-- 9
    second.add( &Primitive( 7 ) );          //   |      +-- Row5
    third.add(  &Primitive( 8 ) );          //   |      |      |
    fourth.add( &Primitive( 9 ) );          //   |      |      +-- 10
    fifth.add(  &Primitive(10 ) );          //   |      +-- 8
    first.traverse();  cout << '\n';        //   +-- 6
}

// Row1  Col2  7  Col3  Row4  9  Row5  10  8  6
```

Example 5:

```
// Purpose.  Composite and Prototype - lightweight persistence

#pragma warning( disable : 4786 )
#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <fstream>
using namespace std;

class Component
{
 public:
   virtual ~Component() { }
   virtual void       traverse() = 0;
   virtual Component* clone() = 0;
   virtual void       initialize( ifstream& ) = 0;
};

namespace Factory
{
   map<string,Component*> hash;
   void add( string s, Component* c ) { hash[s] = c; }
   Component* makeComponent( string name ) { return hash[name]->clone(); }
}

class Leaf : public Component
{
   string value;
public:
   ~Leaf()                     { cout << 'd' << value << ' '; }
```

```
  /*virtual*/ void       traverse() { cout << value << ' '; }
  /*virtual*/ Component* clone()    { return new Leaf(); }
  /*virtual*/ void       initialize( ifstream& is ) { is >> value; }
};

class Composite : public Component
{
  vector<Component*> children;
  string             value;
public:
  ~Composite()
        {
            cout << 'd' << value << ' ';
            for (int i=0; i < children.size(); i++)
                       delete children[i];
        }
  void add( Component* c ) { children.push_back( c ); }
  /*virtual*/ void traverse()
         {
            cout << value << ' ';
            for (int i=0; i < children.size(); i++)
               children[i]->traverse();
         }
  /*virtual*/ Component* clone() { return new Composite(); }
  /*virtual*/ void  initialize( ifstream& is )
        {
            is >> value;
            string str, delim( "/"+value );
            is >> str;
            while (str != delim)
                {
                      add( Factory::makeComponent( str ) );
                      children[children.size()-1]->initialize( is );
                      is >> str;
                }
         }
};

void main( void )
{
  Factory::add( "comp", new Composite() );
  Factory::add( "leaf", new Leaf() );
  ifstream is( "compositeCreate.txt" );
  string str;
  is >> str;
  Component* root = Factory::makeComponent( str );
  root->initialize( is );
  root->traverse();  cout << '\n';
  delete root;       cout << '\n';
}

/***
comp a leaf 1 comp b comp d leaf 8 leaf 9 /d leaf 4 comp e leaf 10
leaf 11 leaf 12 /e leaf 2 comp c leaf 5 leaf 6 leaf 7 /c /b leaf 3 /a
***/
```

```
// a 1 b d 8 9 4 e 10 11 12 2 c 5 6 7 3
// da d1 db dd d8 d9 d4 de d10 d11 d12 d2 dc d5 d6 d7 d3
```

Example 6:

```
// Purpose.  Composite design pattern lab.
//
// Problem.  Approach is now more space efficient, but casting and type
// checking are required to coerce the compiler.  The abstraction needs to
// be improved so that Primitive and Composite can be treated
// transparently in a sibling context, while still maintaining their specialization.
//
// Assignment.
// Create a class Component to serve as a base class.  Primitive and
//   Composite should inherit from Component.
// Move anything that is (or needs to be) common in both Primitive and
//   Composite up into Component.
// Currently class Composite is coupled to itself (the argument to add() and
//   the children private data member).  It needs to be coupled only to its
//   abstract base class.
//  You can now remove: NodeType, reportType(), the casting in main() and
//   Composite::traverse(), and the "type checking" in Composite::traverse()

#include <iostream.h>
enum NodeType { LEAF, INTERIOR };

class Primitive
{
public:
   Primitive( int val ) : value(val), type(LEAF)  { }
   NodeType reportType()                { return type; }
   void     traverse()                  { cout << value << " "; }
private:
   int       value;
   NodeType  type;
};

class Composite
{
public:
   Composite( int val ) : value(val), type(INTERIOR)  { total = 0; }
   NodeType reportType()                          { return type; }
   void     add( Composite* c )                   { children[total++] = c; }
   void traverse()
         {
              cout << value << " ";
              for (int i=0; i < total; i++)
                          if (children[i]->reportType() == LEAF)
                                      ((Primitive*) children[i])->traverse();
                          else
                                      children[i]->traverse();
            }
private:
   int   value;
   NodeType    type;
   int         total;
   Composite*  children[99];
};

void main( void )
{
   Composite  first(1), second(2), third(3);

   first.add( &second );
   first.add( &third );
   first.add( (Composite*) new Primitive(4) );
   second.add( (Composite*) new Primitive(5) );
   second.add( (Composite*) new Primitive(6) );
   third.add( (Composite*) new Primitive(7) );
   first.traverse();
   cout << endl;
}

// 1 2 5 6 3 7 4
```

Decorator

Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Problem

You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

Discussion

Suppose you have a user interface toolkit and you wish to make a border or scrolling feature available to clients without defining new subclasses of all existing classes. The client "attaches" the border or scrolling responsibility to only those objects requiring these capabilities.

```
   Widget*  aWidget = new BorderDecorator(
                       new HorScrollDecorator(
                         new VerScrollDecorator(
                           new TextWidget( 80, 24 ))));
   aWidget->draw();
```

Another example could be cascading responsibilities on to an output stream -

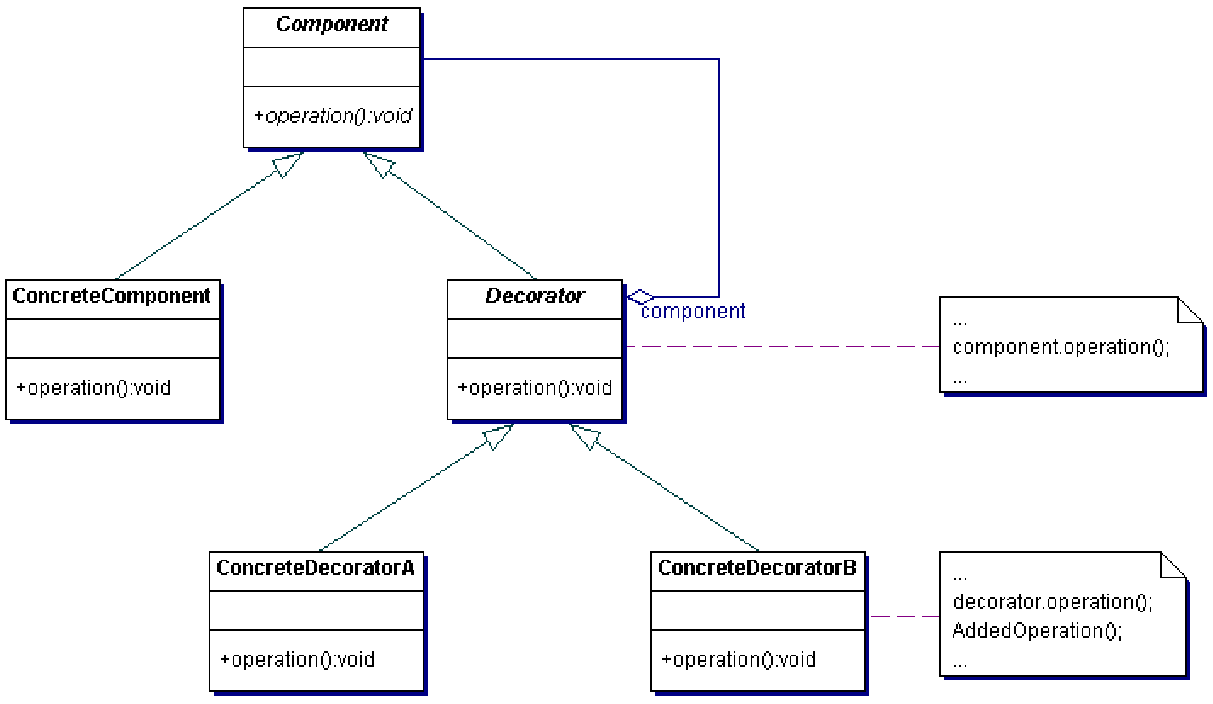
```
   Stream*  aStream = new CompressingStream(
                       new ASCII7Stream(
                         new FileStream( "fileName.dat" )));
   aStream->putString( "Hello world" );
```

The solution to this problem involves encapsulating the original object inside an abstract wrapper interface. Both the decorator objects and the core object inherit from this abstract interface. The interface uses recursive composition to allow an unlimited number of decorator "layers" to be added to each core object.

Note that this pattern allows responsibilities to be added to an object, not methods to an object's interface. The interface presented to the client must remain constant as successive layers are specified.

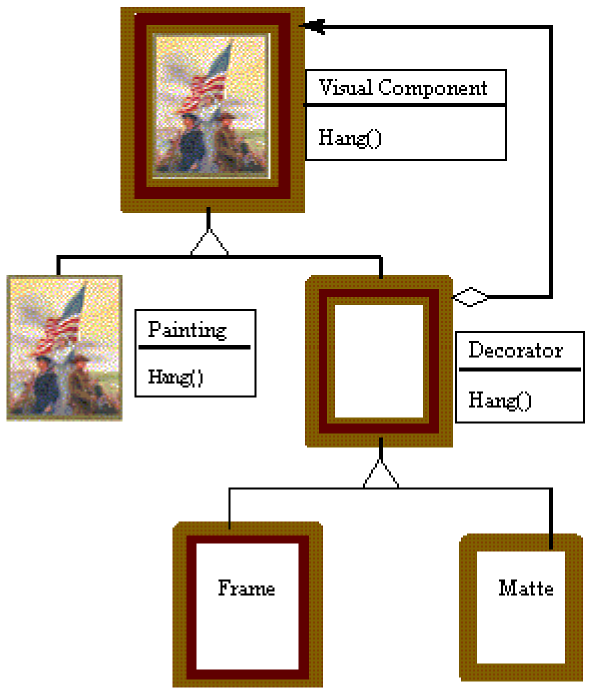
Also note that the core object's identity has now been "hidden" inside of a decorator object. Trying to access the core object directly is now a problem.

Structure



Example

The Decorator attaches additional responsibilities to an object dynamically. The ornaments that are added to pine or fir trees are examples of Decorators. Lights, garland, candy canes, glass ornaments, etc., can be added to a tree to give it a festive look. The ornaments do not change the tree itself which is recognizable as a Christmas tree regardless of particular ornaments used. As an example of additional functionality, the addition of lights allows one to "light up" a Christmas tree. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface. [GOF, p216]

Adapter changes an object's interface, Decorator enhances an object's responsibilities. Decorator is thus more transparent to the client. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters. [GOF, p149]

Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects. [GOF, p219]

A Decorator can be viewed as a degenerate Composite with only one component. However, a Decorator adds additional responsibilities - it isn't intended for object aggregation. [GOF, p184]

Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert. [GOF, p220]

Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition. [GOF, p349]

Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests. [GOF, p220]

Decorator lets you change the skin of an object. Strategy lets you change the guts. [GOF, p184]

Program Examples

Example 1:

```
// Purpose.  Decorator design pattern

// 1. Create a "lowest common denominator" that makes classes interchangeable
// 2. Create a second level base class for optional functionality
// 3. "Core" class and "Decorator" class declare an "isa" relationship
// 4. Decorator class "hasa" instance of the "lowest common denominator"
// 5. Decorator class delegates to the "hasa" object
// 6. Create a Decorator derived class for each optional embellishment
// 7. Decorator derived classes delegate to base class AND add extra stuf
// 8. Client has the responsibility to compose desired configurations

#include <iostream>
using namespace std;

class Widget { public: virtual void draw() = 0; };  // 1. "lowest common denom"

class TextField : public Widget    // 3. "Core" class & "isa"
{
   int width, height;
public:
   TextField( int w, int h ) { width  = w;  height = h; }
   /*virtual*/ void draw() { cout << "TextField: " << width << ", " << height << '\n'; }
};
                                              // 2. 2nd level base class
class Decorator : public Widget               // 3. "isa" relationship
 {
   Widget* wid;                                       // 4. "has a" relationship
public:
   Decorator( Widget* w )  { wid = w; }
   /*virtual*/ void draw() { wid->draw(); }          // 5. Delegation
};
```

```
class BorderDecorator : public Decorator      // 6. Optional embellishment
{
 public:
   BorderDecorator( Widget* w ) : Decorator( w ) { }
   /*virtual*/ void draw()
          {
              Decorator::draw();                              // 7. Delegate to base class
              cout << "   BorderDecorator" << '\n';                    //    and add extra stuff
          }
};

class ScrollDecorator : public Decorator        // 6. Optional embellishment
{
public:
   ScrollDecorator( Widget* w ) : Decorator( w ) { }
   /*virtual*/ void draw()
          {
                 Decorator::draw();                              // 7. Delegate to base class
                 cout << "   ScrollDecorator" << '\n';           //    and add extra stuff
          }
};

void main( void )
{
   // 8. Client has the responsibility to compose desired configurations
   Widget* aWidget = new BorderDecorator(
                     new BorderDecorator(
                        new ScrollDecorator(
                           new TextField( 80, 24 ))));
   aWidget->draw();
}

// TextField: 80, 24
//    ScrollDecorator
//    BorderDecorator
//    BorderDecorator
```

Example 2:

```
// Purpose.  Inheritance run amok

#include <iostream>
using namespace std;

class A
{
public:
   virtual void doIt() { cout << 'A'; }
};

class AwithX : public A
{
   void doX() { cout << 'X';
}
public:
   /*virtual*/ void doIt()
                        {
                              A::doIt();
                              doX();
                        }
 };

class AwithY : public A
{
protected:
   void doY() { cout << 'Y'; }
public:
   /*virtual*/ void doIt()
                        {
                              A::doIt();
                              doY();
                        }
};

class AwithZ : public A
{
protected:
   void doZ() { cout << 'Z'; }
public:
   /*virtual*/ void doIt()
                        {
                              A::doIt();
                              doZ();
                        }
};

class AwithXY : public AwithX, public AwithY
{
 public:
   /*virtual*/ void doIt()
                        {
                              AwithX::doIt();
                              AwithY::doY();
                        }
};

class AwithXYZ : public AwithX, public AwithY, public AwithZ
{
 public:
   /*virtual*/ void doIt()
                        {
                              AwithX::doIt();
                              AwithY::doY();
                              AwithZ::doZ();
                        }
};
```

```
void main( void )
{
   AwithX    anX;
   AwithXY   anXY;
   AwithXYZ  anXYZ;
   anX.doIt();    cout << '\n';
   anXY.doIt();   cout << '\n';
   anXYZ.doIt();  cout << '\n';
}

// AX
// AXY
// AXYZ
```

Example 3:

```
// Purpose.  Replacing inheritance with wrapping-delegation
//
// Discussion.  Use aggregation instead of inheritance to implement
// embellishments to a "core" object.  Client can dynamically compose
// permutations, instead of the architect statically wielding multiple inheritance.

#include <iostream>
using namespace std;

class I
{
 public:
   virtual ~I() { }
   virtual void doIt() = 0;
};

class A : public I
{
 public:
   ~A() { cout << "A dtor" << '\n'; }
   /*virtual*/ void doIt() { cout << 'A'; }
};

class D : public I
{
   I* wrappee;
public:
   D( I* inner )           { wrappee = inner; }
   ~D()                    { delete wrappee; }
   /*virtual*/ void doIt() { wrappee->doIt(); }
};

class X : public D
{
 public:
   X( I* core ) : D(core) { }
   ~X() { cout << "X dtor" << "   "; }
   /*virtual*/ void doIt() { D::doIt();  cout << 'X'; }
};
```

```
class Y : public D
{
 public:
   Y( I* core ) : D(core) { }
   ~Y() { cout << "Y dtor" << "   "; }
   /*virtual*/ void doIt() { D::doIt();  cout << 'Y'; }
};

class Z : public D
{
 public:
   Z( I* core ) : D(core) { }
   ~Z() { cout << "Z dtor" << "   "; }
   /*virtual*/ void doIt() { D::doIt();  cout << 'Z'; }
};

void main( void )
{
   I* anX = new X( new A );
   I* anXY = new Y( new X( new A ) );
   I* anXYZ = new Z( new Y( new X( new A ) ) );
   anX->doIt();    cout << '\n';
   anXY->doIt();   cout << '\n';
   anXYZ->doIt();  cout << '\n';
   delete anX;   delete anXY;   delete anXYZ;
}

// AX
// AXY
// AXYZ
// X dtor   A dtor
// Y dtor   X dtor   A dtor
// Z dtor   Y dtor   X dtor   A dtor
```

Example 4:

```
// Purpose.  Decorator - encoding and decoding layers of header/packet/trailer
#include <iostream>
#include <string>
using namespace std;

class Interface { public:
   virtual ~Interface() { }
   virtual void write( string& ) = 0;
   virtual void read(  string& ) = 0;
};

class Core : public Interface { public:
   ~Core() { cout << "dtor-Core\n"; }
   /*virtual*/ void write( string& b ) { b += "MESSAGE|"; }
   /*virtual*/ void read( string& );
};
```

```
class Decorator : public Interface
 {
   Interface* inner;
public:
   Decorator( Interface* c ) { inner = c; }
   ~Decorator()              { delete inner; }
   /*virtual*/ void write( string& b ) { inner->write( b ); }
   /*virtual*/ void read(  string& b ) { inner->read( b ); }
};

class Wrapper : public Decorator
{
   string forward, backward;
public:
   Wrapper( Interface* c, string str ) : Decorator(c)
               {
                    forward = str;
                     string::reverse_iterator it;
                     it = str.rbegin();
                     for ( ; it != str.rend(); ++it)
                                   backward += *it;
                }
   ~Wrapper() { cout << "dtor-" << forward << "  "; }
   void write( string& );
   void read(  string& );
};

void main( void )
{
   Interface* object = new Wrapper( new Wrapper( new Wrapper(
                        new Core(), "123" ), "abc" ), "987" );
   string buf;
   object->write( buf );
   cout << "main: " << buf << endl;
   object->read( buf );
   delete object;
}

// main: 987]abc]123]MESSAGE|321]cba]789]
// Wrapper: 987
// Wrapper: abc
// Wrapper: 123
// Core: MESSAGE
// Wrapper: 321
// Wrapper: cba
// Wrapper: 789
// dtor-987  dtor-abc  dtor-123  dtor-Core

void Core::read(string& b)
{
   int num = b.find_first_of( '|' );
   cout << "Core: " << b.substr(0,num) << '\n';
   b = b.substr(num+1);
}
```

```
void Wrapper::write( string& b )
{
  b += forward + "]";
  Decorator::write( b );
  b += backward + "]";
}

void Wrapper::read( string& b )
{
  int num = b.find_first_of( ']' );
  cout << "Wrapper: " << b.substr(0,num) << '\n';
  b = b.substr(num+1);
  Decorator::read( b );
  num = b.find_first_of( ']' );
  cout << "Wrapper: " << b.substr(0,num) << '\n';
  b = b.substr(num+1);
}
```

Example 5:

```
// Purpose.  Decorator design pattern lab
//
// Problem.  Inheritance is being used to produce lots of incremental
// customizations.  This is fine - until all the potential permutations of
// options gets out of hand.  Another limitation is the static nature of
// inheritance.  For any particular combination of options to be useable,
// the class hierarchy must explicitly implement that combination.  It would
// be much better if all the individual "options" could each be represented
// by their own class, and, the client could dynamically compose her
// peculiar list of options at run-time.
//
// Assignment.
// o Add an abstract base class Expression with the single pure virtual
//   function evaluate().
// o Function should inherit from Expression (but otherwise remain unchanged).
// o Add a base class Decorator that inherits from Expression.
// o Add an Expression* data member to Decorator (this will be used to remember
//   the "core" object that this Decorator instance is "wrapping").
// o Decorator needs a constructor to initialize this Expression* data member.
// o Decorator's evaluate() should do nothing more than delegate to the
//   Expression* data member.
// o LowCut and HighCut should now inherit from Decorator instead of Function.
// o Eliminate MiddlePass.
// o Add an Expression* parameter to LowCut's and HighCut's constructors (this
//   is the "core" object that this "onion skin" is going to be wrapping).
// o Use a member initialization list to communicate the Expression* parameter
//   to the Decorator base class.
// o LowCut's and HighCut's evaluate() should now call base class Decorator's
//   evaluate() instead of Function's (Function is no longer serving as a base
//   class).
// o Now that LowCut and HighCut no longer have an "isa" relationship to
//   Function (they are now "embellishments" for a core object of type
//   Function), the "lc" and "hc" local variables in main() must have a
//   Function* at the "core" of their constructor arguments.  This could be
//   done with an inline dynamic allocation (new Function).
// o The MiddlePass class no longer exists.  Therefore, the "mp" local variable
//   in main() will be configured by using a layering of "decorator" objects on
//   a "core" object.

#include <iostream.h>
#include <stdlib.h>
#include <time.h>

class Function
 {
public:
   Function()
         {
              time_t t;
              srand((unsigned) time(&t));
          }
   virtual int evaluate()
         {
              return rand() % 30;
         }
};

class LowCut : virtual public Function
{
public:
   LowCut( int min )
         {
              minimum_ = min;
         }
   int evaluate()
         {
              int temp = Function::evaluate();
              if (temp < minimum_)
                            return minimum_;
              return temp;
         }
protected:
   int  minimum_;
};

class HighCut : virtual public Function
{
public:
   HighCut( int max )
         {
              maximum_ = max;
         }
   int evaluate()
          {
              int temp = Function::evaluate();
              if (temp > maximum_)
                            return maximum_;
              return temp;
          }
protected:
  int  maximum_;
};

class MiddlePass : public LowCut, public HighCut
{
public:
  MiddlePass( int min, int max ) : LowCut(min), HighCut(max) { }
  int evaluate()
        {
            int temp = LowCut::evaluate();
            if (temp > maximum_) return maximum_;
                    return temp;
        }
};

void main( void )
{
  LowCut      lc( 7 );
  HighCut     hc( 23 );
  MiddlePass  mp( 10, 20 );
  int   i;

  for (i=0; i < 25; i++) cout << lc.evaluate() << " "; cout << endl;
  for (i=0; i < 25; i++) cout << hc.evaluate() << " "; cout << endl;
  for (i=0; i < 25; i++) cout << mp.evaluate() << " "; cout << endl;
}

// 27 18 7 13 19 13 22 16 22 16 17 11 13 13 10 20 22 7 8 23 14 7 29 7 15
// 2 22 23 15 8 16 7 18 7 10 6 21 6 19 3 23 0 20 5 13 23 3 2 6 1
// 18 19 20 20 12 10 14 14 13 14 20 20 20 10 12 18 10 11 10 18 20 20 10 10 10
//
// 11 19 26 27 7 7 25 15 10 9 23 8 10 13 20 18 14 7 7 17 7 7 7 22 25
// 11 4 6 23 22 10 15 23 22 2 12 19 19 12 13 18 6 15 6 1 13 1 10 14 23
// 18 20 20 20 10 20 17 10 20 15 20 17 10 16 10 10 10 17 20 20 11 14 20 10 10
//
// 7 7 11 29 27 7 11 16 27 15 10 26 27 7 11 28 17 7 7 13 11 20 26 21 29
// 22 23 23 23 3 13 17 13 12 0 23 23 3 11 14 23 13 4 0 23 9 22 23 14 14
// 10 10 10 10 10 10 10 10 19 14 14 10 13 10 11 14 20 16 20 20 20 10 18 18 10
```

Facade

Intent

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Problem

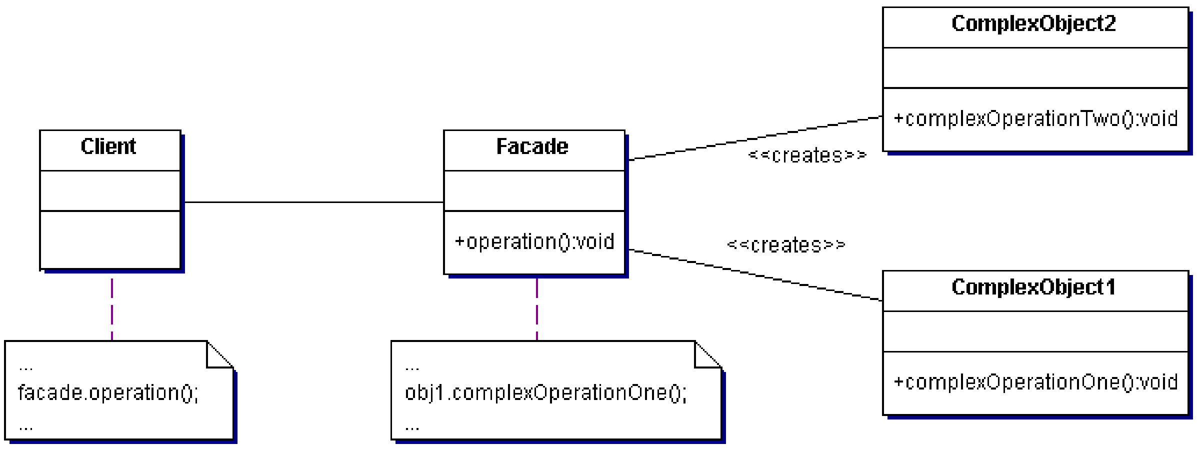
A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

Discussion

Facade discusses encapsulating a complex subsystem within a single interface object. This reduces the learning curve necessary to successfully leverage the subsystem. It also promotes decoupling the subsystem from its potentially many clients. On the other hand, if the Facade is the only access point for the subsystem, it will limit the features and flexibility that "power users" may need.

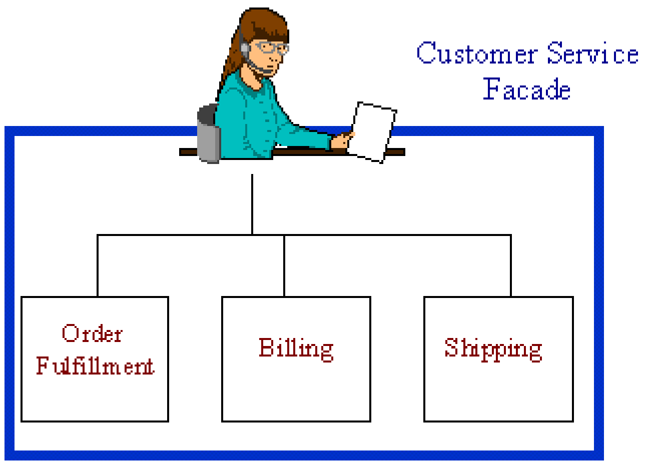
The Facade object should be a fairly simple advocate or facilitator. It should not become an all-knowing oracle or "god" object.

Structure



Example

The Facade defines a unified, higher level interface to a subsystem that makes it easier to use. Consumers encounter a Facade when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a Facade, providing an interface to the order fulfillment department, the billing department, and the shipping department. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

Facade defines a new interface, whereas Adapter uses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one. [GOF, p219]

Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem. [GOF, p138]

Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communications between colleague objects. It routinely "adds value", and it is known/referenced by the colleague objects. In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes. [GOF. p193]

Abstract Factory can be used as an alternative to Facade to hide platform-specific classes. [GOF, p193]

Facade objects are often Singletons because only one Facade object is required. [GOF, p193]

Program Examples

Example 1:

```
// Purpose.  Facade
//
// Discussion.  Class Compute models a decimal digit adder module.  An
// entire "subsystem" can be configured by linking as many of these
// modules as the desired precision requires.  The "subsystem" being
// modeled in main() is complex and  burdensome to use.  Wrapping this
// subsystem inside of a Facade that exports a simple interface is desirable.

#include <iostream.h>
#include <string.h>
#define sl strlen

class Compute
{
public:
char add( char a, char b, int& c)
        {
                int result = a + b + c - 96;
                c = 0;
                if (result > 9)
                        {
                                result -= 10;
                                c = 1;
                        }
                return result + 48;
        }
};

void main( void )
{
Compute  tens, ones;
char     a[9], b[9], c, d;
int      cary;
while (1)
        {
                cout << "Enter 2 nums: ";
                cin >> a >> b;
                cout << "   sum is ";
                cary = 0;
                if ((sl(a) > 1) && (sl(b) > 1))
                        {
                                c = ones.add( a[1], b[1], cary);
                                d = tens.add( a[0], b[0], cary);
                        }
                 else if (sl(a) > 1)
                        {
```

```
                        c = ones.add( a[1], b[0], cary);
                        d = tens.add( a[0],  '0', cary);
                    }
                else if (sl(b) > 1)
                    {
                        c = ones.add( b[1], a[0], cary);
                        d = tens.add( b[0],  '0', cary);
                    }
                else
                     {
                        c = tens.add( a[0], b[0], cary);
                        d = 'x';
                    }
                if (cary)
                     cout << '1';
                if (d != 'x')
                    cout << d;
                cout << c << endl;
        }
}

// Enter 2 nums: 99 99
//    sum is 198
// Enter 2 nums: 38 83
//    sum is 121
// Enter 2 nums: 5 6
//    sum is 11
```

Example 2:

```
#include <iostream.h>
#include <string.h>
#define sl strlen

class Compute
{
public:
  char add( char a, char b, int& c )
        {
                int result = a + b + c - 96;"
                c = 0;
                  if (result > 9)
                    {
                      result -= 10;
                      c = 1;
                    }
                return result + 48;
         }
};

class Facade
{
public:
        char* add( char* a, char* b )
        {
                int cary = 0, i = 0;
                char c, d;
                if ((sl(a) > 1) && (sl(b) > 1))
                {
                        c = ones.add( a[1], b[1], cary );
                        d = tens.add( a[0], b[0], cary );
                }
                else if (sl(a) > 1)
                {
                        c = ones.add( a[1], b[0], cary );
                           d = tens.add( a[0],  '0', cary );
                }
                else if (sl(b) > 1)
                {
                        c = ones.add( b[1], a[0], cary );
                        d = tens.add( b[0],  '0', cary );
                 }
                else
                 {
                         c = tens.add( a[0], b[0], cary );
                           d = 'x';
                 }
                  if (cary)
                        ans[i++] = '1';
                  if (d != 'x')
                        ans[i++] =  d;
                 ans[i++] = c;
                    ans[i] = '\0';
                return ans;
        }
private:
        Compute  tens, ones;
         char     ans[9];
};

void main( void )
{
        Facade  f;
        char    a[9], b[9];
                while (1)
                {
                   cout << "Enter 2 nums: ";
                   cin >> a >> b;
                   cout <<"   sum is "<< f.add(a,b)<< endl;
                }
}

// Enter 2 nums: 9 13
//    sum is 22
// Enter 2 nums: 19 8
//    sum is 27
// Enter 2 nums: 3 99
//    sum is 102
```

Example 3:

```
// Purpose.  Facade design pattern demo.
//
// Discussion.  Structuring a system into subsystems helps reduce
// complexity.  A common design goal is to minimize the communication and
// dependencies between subsystems.  One way to achieve this goal is to
// introduce a "facade" object that provides a single, simplified
// interface to the many, potentially complex, individual interfaces
// within the subsystem.  In this example, the "subsystem" for responding
// to a networking service request has been modeled, and a facade
// (FacilitiesFacade) interposed.  The facade "hides" the twisted and
// bizarre choreography necessary to satisfy even the most basic of
// requests.  All the user of the facade object has to do is make one or
// two phone calls a week for 5 months, and a completed service request results.

#include <iostream.h>

class MisDepartment
{
public:
        void submitNetworkRequest() { _state = 0; }
        bool checkOnStatus()
        {
                _state++;
                if (_state == Complete)
                        return 1;
                return 0;
        }
private:
        enum States {Received, DenyAllKnowledge, ReferClientToFacilities,
                FacilitiesHasNotSentPaperwork, ElectricianIsNotDone,
                ElectricianDidItWrong, DispatchTechnician, SignedOff, DoesNotWork,
                FixElectriciansWiring, Complete};
        int _state;
};

class ElectricianUnion
{
public:
        void submitNetworkRequest() { _state = 0; }
        bool checkOnStatus()
                {
                        _state++;
                        if (_state == Complete)
                                return 1;
                        return 0;
                }
private:
        enum States {Received, RejectTheForm, SizeTheJob, SmokeAndJokeBreak,
                WaitForAuthorization, DoTheWrongJob, BlameTheEngineer,
            WaitToPunchOut,DoHalfAJob, ComplainToEngineer, GetClarification,
            CompleteTheJob,TurnInThePaperwork, Complete};
   int _state;
};
```

```
class FacilitiesDepartment
{
public:
        void submitNetworkRequest() { _state = 0; }
        bool checkOnStatus()
        {
                _state++;
                if (_state == Complete)
                        return 1;
                return 0;
        }
private:
        enum States {Received, AssignToEngineer, EngineerResearches,
                RequestIsNotPossible, EngineerLeavesCompany,
AssignToNewEngineer,
                NewEngineerResearches, ReassignEngineer, EngineerReturns,
                EngineerResearchesAgain, EngineerFillsOutPaperWork, Complete};
        int _state;
};

class FacilitiesFacade
{
public:
        FacilitiesFacade()      { _count = 0; }
        void submitNetworkRequest() { _state = 0; }
        bool checkOnStatus()
        {
                _count++;
                /* Job request has just been received */
                if (_state == Received)
                {
                        _state++;
                        /* Forward the job request to the engineer */
                        _engineer.submitNetworkRequest();
                        cout << "submitted to Facilities - " << _count
                                << " phone calls so far" << endl;
                }
                else if (_state == SubmitToEngineer)
                {
                        /* If engineer is complete, forward to electrician */
                        if (_engineer.checkOnStatus())
                        {
                                _state++;
                                _electrician.submitNetworkRequest();
                                cout << "submitted to Electrician - " << _count
                                        << " phone calls so far" << endl;
                        }
                 }
                else if (_state == SubmitToElectrician)
                {
                        /* If electrician is complete, forward to technician */
                        if (_electrician.checkOnStatus())
                        {
                                _state++;
                        _technician.submitNetworkRequest();
                        cout << "submitted to MIS - " << _count
                            << " phone calls so far" << endl;
                    }
                }
                else if (_state == SubmitToTechnician)
                {
                    /* If technician is complete, job is done */
                    if (_technician.checkOnStatus())
                        return 1;
                }
                /* The job is not entirely complete */
                return 0;
            }
    int getNumberOfCalls() { return _count; }
    private:
            enum States {Received, SubmitToEngineer, SubmitToElectrician,
                SubmitToTechnician};
            int              _state;
            int              _count;
            FacilitiesDepartment  _engineer;
            ElectricianUnion      _electrician;
            MisDepartment         _technician;
    };

    void main()
    {
            FacilitiesFacade  facilities;

            facilities.submitNetworkRequest();
            /* Keep checking until job is complete */
            while ( ! facilities.checkOnStatus())
                ;
            cout << "job completed after only " << facilities.getNumberOfCalls()<< "
    phone calls" << endl;
    }

    // submitted to Facilities - 1 phone calls so far
    // submitted to Electrician - 12 phone calls so far
    // submitted to MIS - 25 phone calls so far
    // job completed after only 35 phone calls
```

Flyweight

Intent

Use sharing to support large numbers of fine-grained objects efficiently.

Problem

Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage.

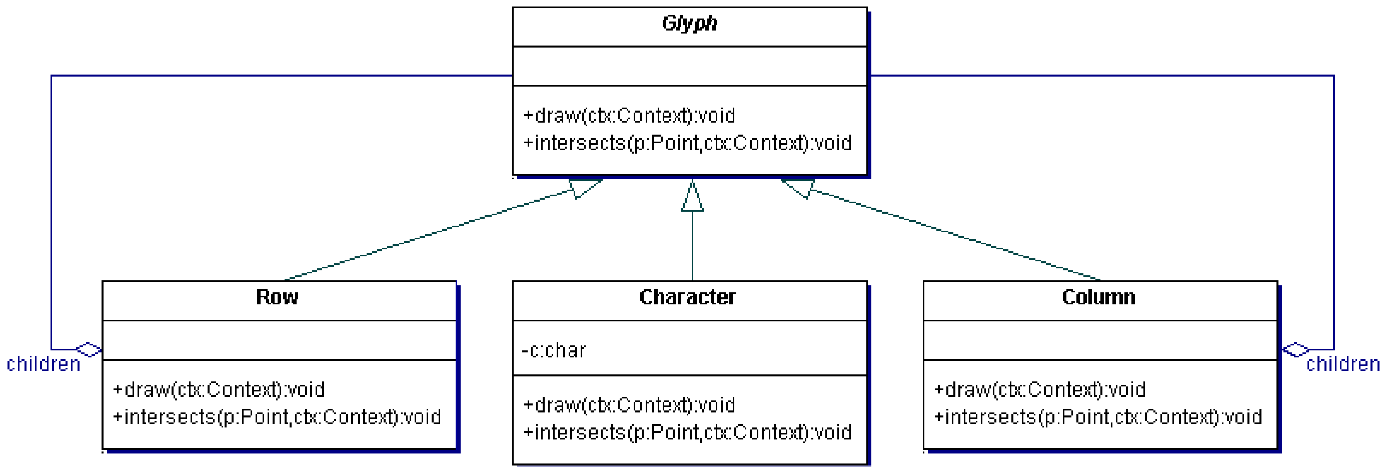
Discussion

The Flyweight pattern describes how to share objects to allow their use at fine granularities without prohibitive cost. Each "flyweight" object is divided into two pieces: the state-dependent (extrinsic) part, and the state-independent (intrinsic) part. Intrinsic state is stored (shared) in the Flyweight object. Extrinsic state is stored or computed by client objects, and passed to the Flyweight when its operations are invoked.

An illustration of this approach would be Motif widgets that have been re-engineered as light-weight gadgets. Whereas widgets are "intelligent" enough to stand on their own; gadgets exist in a dependent relationship with their parent layout manager widget. Each layout manager provides context-dependent event handling, real estate management, and resource services to its flyweight gadgets, and each gadget is only responsible for context-independent state and behavior.

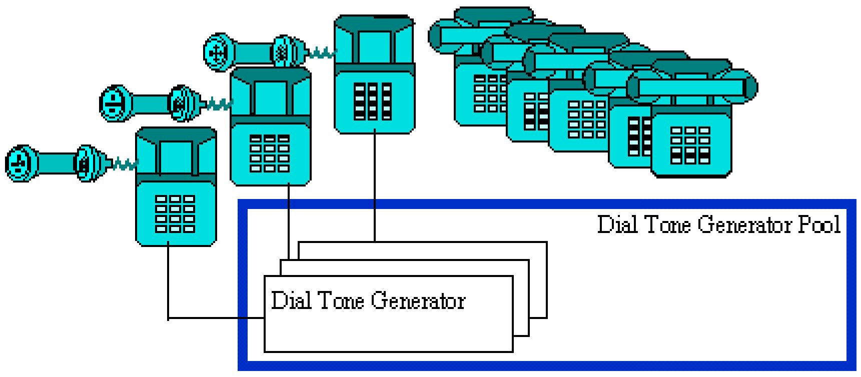
Structure

Note that *Row* and *Column* are playing the role of *Composite* in the Composite pattern. The *Context* parameters are providing the requisite extrinsic state to each method. No "factory" is modeled here for serving up flyweight objects.


Glyph
+draw(ctx:Context):void
+intersects(p:Point,ctx:Context):void
Row
+draw(ctx:Context):void
+intersects(p:Point,ctx:Context):void
Character
-c:char
+draw(ctx:Context):void
+intersects(p:Point,ctx:Context):void
Column
+draw(ctx:Context):void
+intersects(p:Point,ctx:Context):void
children
children

Example

The Flyweight uses sharing to support large numbers of objects efficiently. The public switched telephone network is an example of a Flyweight. There are several resources such as dial tone generators, ringing generators, and digit receivers that must be shared between all subscribers. A subscriber is unaware of how many resources are in the pool when he or she lifts the handset to make a call. All that matters to subscribers is that a dial tone is provided, digits are received, and the call is completed. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem. [GOF, p138]

Flyweight is often combined with Composite to implement shared leaf nodes. [GOF, p206]

Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight. [GOF. p255]

Flyweight explains when and how State objects can be shared. [GOF, p313]

Program Examples

Example 1:

```
// Purpose.  Flyweight
//
// Discussion.  Trying to use objects at very low levels of granularity
// is nice, but the overhead may be prohibitive.  Flyweight suggests
// removing the non-shareable state from the class, and having the cli-
// ent supply it when methods are called.  This places more respon-
// sibility on the client, but, considerably fewer instances of the
// Flyweight class are now created.
// Sharing of these instances is facilitated by introducing a Factory
// class that maintains a "cache" of existing Flyweights.
//
// In this example, the "X" state is considered shareable (within each
// row anyways), and the "Y" state has been externalized (it is supplied
// by the client when report() is called).

#include <iostream.h>
const int X = 6;
const int Y = 10;

class Gazillion
{
public:
Gazillion()
         {
                 val1_ = num_ / Y;
                 val2_ = num_ % Y;
                 num_++;
         }
void report()
         {
                 cout << val1_ << val2_ << ' ';
         }
private:
         int    val1_;
         int    val2_;
         static int num_;
};
```

```
int Gazillion::num_ = 0;

void main( void )
{
Gazillion  matrix[X][Y];
for (int i=0; i < X; i++)
        {
                for (int j=0; j < Y; j++)
                matrix[i][j].report();
                cout << endl;
        }
}

// 00 01 02 03 04 05 06 07 08 09
// 10 11 12 13 14 15 16 17 18 19
// 20 21 22 23 24 25 26 27 28 29
// 30 31 32 33 34 35 36 37 38 39
// 40 41 42 43 44 45 46 47 48 49
// 50 51 52 53 54 55 56 57 58 59
```

Example 2:

```
#include <iostream.h>

const int X = 6;
const int Y = 10;

class Gazillion
{
public:
Gazillion( int in )
        {
        val1_ = in;
        cout << "ctor: "<< val1_<<endl;
        }
~Gazillion()
{
        cout << val1_ << ' ';
}
void report( int in )
{
        cout << val1_ << in << ' ';
}
private:
        int  val1_;
};

class Factory
{
public:
static Gazillion* getFly(int in)
        {
                if ( ! pool_[in])
                pool_[in] =new Gazillion( in );
                return pool_[in];
        }
static void cleanUp()
        {
        cout << "dtors: ";
        for (int i=0; i < X; i++)
                if (pool_[i])
                        delete pool_[i];
                cout << endl;
        }
private:
        static Gazillion*  pool_[X];
};

Gazillion*  Factory::pool_[]  = {0,0,0,0,0,0 };

void main( void )
{
for (int i=0; i < X; i++)
        {
                for (int j=0; j < Y; j++)
                Factory::getFly(i)->report(j);
                cout << endl;
        }
Factory::cleanUp();
}

// ctor: 0
// 00 01 02 03 04 05 06 07 08 09
// ctor: 1
// 10 11 12 13 14 15 16 17 18 19
// ctor: 2
// 20 21 22 23 24 25 26 27 28 29
// ctor: 3
// 30 31 32 33 34 35 36 37 38 39
// ctor: 4
// 40 41 42 43 44 45 46 47 48 49
// ctor: 5
// 50 51 52 53 54 55 56 57 58 59
// dtors: 0 1 2 3 4 5
```

Example 3:

```
// Purpose.  Flyweight design pattern demo.
//
// Discussion.  Flyweight describes how to share objects, so that their
// use at fine granularities is not cost prohibitive.  A key concept is
// the distinction between "intrinsic" and "extrinsic" state.  Intrinsic
// state consists of information that is independent of the flyweight's
// context - information that is sharable (i.e. each Icon's name, width,
// and height).  It is stored in the flyweight (i.e. the Icon class).
// Extrinsic state cannot be shared, it depends on and varies with the
// flyweight's context (i.e. the x,y position that each Icon instance's
// upper left corner will be drawn at).  Extrinsic state is stored or
// computed by the client and is passed to the flyweight when an operation
// is invoked.  Clients should not instantiate Flyweights directly, they
// should obtain them exclusively from a FlyweightFactory object to ensure
// they are shared properly.

#include <iostream.h>
#include <string.h>

class Icon
{
public:
        Icon( char* fileName )
        {
                strcpy( _name, fileName );
                if ( ! strcmp(fileName, "go"))     { _width = 20;  _height = 20; }
                if ( ! strcmp(fileName, "stop"))   { _width = 40;  _height = 40; }
                if ( ! strcmp(fileName, "select")) { _width = 60;  _height = 60; }
                if ( ! strcmp(fileName, "undo"))   { _width = 30;  _height = 30; }
        }
        const char* getName() { return _name; }
        void draw( int x, int y )
        {
                cout << "   drawing " << _name << ": upper left (" << x << "," << y
                << ") - lower right (" << x + _width << "," << y + _height << ")"
                << endl;
        }
private:
        char  _name[20];
        int   _width;
        int   _height;
};

class FlyweightFactory
{
public:
        static Icon* getIcon( char* name )
         {
                for (int i=0; i < _numIcons; i++)
                        if ( ! strcmp( name, _icons[i]->getName() ))
                                        return _icons[i];
                _icons[_numIcons] = new Icon( name );
                return _icons[_numIcons++];
         }
        static void reportTheIcons()
        {
                cout << "Active Flyweights: ";
                for (int i=0; i < _numIcons; i++)
                        cout << _icons[i]->getName() << " ";
                cout << endl;
        }
private:
        enum { MAX_ICONS = 5 };
        static int    _numIcons;
        static Icon*  _icons[MAX_ICONS];
};
```

```
int   FlyweightFactory::_numIcons = 0;
Icon* FlyweightFactory::_icons[];


class DialogBox
{
public:
        DialogBox( int x, int y, int incr ) : _iconsOriginX(x), _iconsOriginY(y),
        _iconsXIncrement(incr) { }
        virtual void draw() = 0;
protected:
        Icon* _icons[3];
        int   _iconsOriginX;
        int   _iconsOriginY;
        int   _iconsXIncrement;
};

class FileSelection : public DialogBox
{
public:
        FileSelection( Icon* first, Icon* second, Icon* third ) :
                DialogBox(100,100,100)
                {
                _icons[0] = first;
                _icons[1] = second;
                _icons[2] = third;
                 }
        void draw()
        {
                cout << "drawing FileSelection:" << endl;
                for (int i=0; i < 3; i++)
                        _icons[i]->draw( _iconsOriginX + (i * _iconsXIncrement),
                                _iconsOriginY );
         }
};

class CommitTransaction : public DialogBox
{
public:
        CommitTransaction( Icon* first, Icon* second, Icon* third ) :
                DialogBox(150,150,150)
                {
                _icons[0] = first;
                _icons[1] = second;
                _icons[2] = third;
                }
        void draw()
                {
                cout << "drawing CommitTransaction:" << endl;
                for (int i=0; i < 3; i++)
                        _icons[i]->draw( _iconsOriginX + (i * _iconsXIncrement),
                                _iconsOriginY );
                 }
};
```

```
void main()
{
        DialogBox* dialogs[2];
        dialogs[0] = new FileSelection(
                FlyweightFactory::getIcon("go"),
                FlyweightFactory::getIcon("stop"),
                FlyweightFactory::getIcon("select") );
        dialogs[1] = new CommitTransaction(
                FlyweightFactory::getIcon("select"),
                FlyweightFactory::getIcon("stop"),
                FlyweightFactory::getIcon("undo") );

        for (int i=0; i < 2; i++)
                dialogs[i]->draw();

        FlyweightFactory::reportTheIcons();
}

// drawing FileSelection:
//    drawing go: upper left (100,100) - lower right (120,120)
//    drawing stop: upper left (200,100) - lower right (240,140)
//    drawing select: upper left (300,100) - lower right (360,160)
//    drawing CommitTransaction:
//    drawing select: upper left (150,150) - lower right (210,210)
//    drawing stop: upper left (300,150) - lower right (340,190)
//    drawing undo: upper left (450,150) - lower right (480,180)
// Active Flyweights: go stop select undo
```

Example 4:

```
// Purpose.  Flyweight design pattern lab
//
// Discussion.  Let's say 20 bytes per gadget is unacceptable overhead.  We
// want to "lighten up" the Gadget hierarchy, and, share a small number of
// "real" instances across a larger number of "virtual" instances.
//
// Assignment.
// o Replace the current main() with the one commented-out at the end
// o Gadget's x_, y_, and w_ are not shareable.  Remove them from Gadget's
//   private state, and, modify draw() to accept x, y, and w.
// o Add a Factory class that supports the expectations in the new main()
// o Factory could use a static array of 2 Gadget ptrs as its "cache" or
//   "pool" data structure (don't forget to init this static array)
// o getFlyweight() will check this static array to see if the requested
//   flyweight exists (if it doesn't, it will create it), and then return it
// o Notice how the "client" code in main() got more complicated.  The client
//   is responsible for knowing (or computing) the new "extrinsic" state and
//   passing that state to the flyweight objects.

#include <iostream.h>

// Microsoft Visual C++ code
#include <Wtypes.h>
```

```
#include <wincon.h>
HANDLE  console_ = GetStdHandle(STD_OUTPUT_HANDLE);
void gotoxy( int x, int y )
{
   COORD  coord;
   coord.X = x-1;
   coord.Y = y-1;
   SetConsoleCursorPosition( console_, coord );
}

class Gadget
{
public:
   Gadget( int col, int row, int width, char ch )
   {
      x_ = col;  y_ = row;  w_ = width;
      ch_ = ch;
   }
   virtual void draw()
{
      int  i;
      gotoxy( x_, y_ );
      for (i=0; i < w_; i++)    cout << ch_;
      cout << endl;
      gotoxy( x_, y_+1 );
      cout << ch_;
      for (i=0; i < w_-2; i++)  cout << ' ';
      cout << ch_ << endl;
      gotoxy( x_, y_+2 );
      for (i=0; i < w_; i++)    cout << ch_;
      cout << endl;
 }
private:
   long  x_, y_, w_;
   char  ch_;
};

class Label : public Gadget
{
public:
   Label( int col, int row, int width ) : Gadget(col,row,width,'@') { }
};

class Edit : public Gadget
{
public:
   Edit( int col, int row, int width ) : Gadget(col,row,width,'#') { }
};

void main( void )
{
   Gadget* list[] = {new Label( 10, 1, 5 ), new Label( 10, 4, 7 ), new Label( 10, 7, 9 ),
                              new Edit( 16, 1, 9 ),  new Edit( 18, 4, 7 ),  new Edit( 20, 7, 5 )
};
for (int i=0; i < 6; i++)
            list[i]->draw();
  for (i=0; i < 6; i++)
            delete list[i];
  cout << "\nsizeof Gadget is " << sizeof(Gadget) << endl;
  cout << "sizeof Label is " << sizeof(Label) << endl;
}

//          @@@@@ #########
//          @   @ #       #
//          @@@@@ #########
//          @@@@@@@ #######
//          @     @ #     #
//          @@@@@@@ #######
//          @@@@@@@@@ #####
//          @       @ #   #
//          @@@@@@@@@ #####
//
// sizeof Gadget is 20
// sizeof Label is 20

#if 0
void main( void )
{
  Gadget*  list[6];
  list[0] = Factory::getFlyweight( "label" );
  list[1] = Factory::getFlyweight( "label" );
  list[2] = Factory::getFlyweight( "label" );
  list[3] = Factory::getFlyweight( "edit" );
  list[4] = Factory::getFlyweight( "edit" );
  list[5] = Factory::getFlyweight( "edit" );
  for (int i=0; i < 3; i++)
        list[i]->draw( 10, i*3+1, i*2+5 );
  for (i=3; i < 6; i++)
    list[i]->draw( (i-3)*2+16, (i-3)*3+1, 9-(i-3)*2 );
  cout << "\nsizeof Gadget is " << sizeof(Gadget) << endl;
  cout << "sizeof Label is " << sizeof(Label) << endl;
}
#endif
```

Proxy

Intent

Provide a surrogate or placeholder for another object to control access to it.

Problem

You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.

Discussion

Design a surrogate, or proxy, object that: instantiates the real object the first time the client makes a request of the proxy, remembers the identity of this real object, and forwards the instigating request to this real object. Then all subsequent requests are simply forwarded directly to the encapsulated real object.

There are four common situations in which the Proxy pattern is applicable.

A virtual proxy is a placeholder for "expensive to create" objects. The real object is only created when a client first requests/accesses the object.

A remote proxy provides a local representative for an object that resides in a different address space. This is what the "stub" code in RPC and CORBA provides.

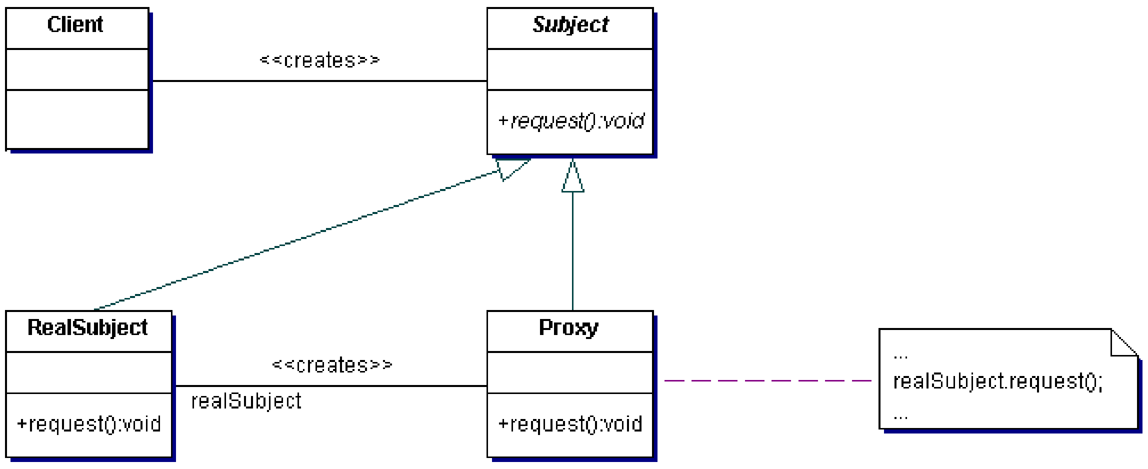
A protective proxy controls access to a sensitive master object. The "surrogate" object checks that the caller has the access permissions required prior to forwarding the request.

A smart proxy interposes additional actions when an object is accessed. Typical uses include:

- Counting the number of references to the real object so that it can be freed automatically when there are no more references (aka smart pointer),
- Loading a persistent object into memory when it's first referenced,
- Checking that the real object is locked before it is accessed to ensure that no other object can change it.

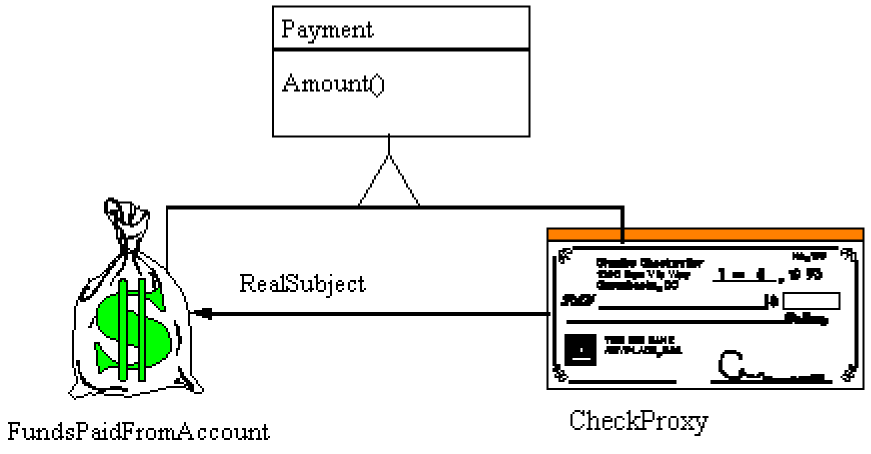
Structure

This diagram is from javacoder.net.



Example

The Proxy provides a surrogate or place holder to provide access to an object. A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface. [GOF. p216]

Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests. [GOF, p220]

Program Examples

Example 1:

```
// Purpose.  Direct coupling, lots of start-up and shut-down overhead

#include <iostream>
#include <string>
using namespace std;

class Image
{
   int        id;
   static int next;
public:
   Image() { id = next++;  cout << "   $$ ctor: "<< id << '\n'; }
   ~Image()              { cout << "   dtor: " << id << '\n'; }
   void draw()           { cout << "   drawing image " << id << '\n'; }
};
int Image::next = 1;

void main( void )
{
   Image images[5];
   int   i;

   cout << "Exit[0], Image[1-5]: ";
   cin >> i;
   while (i)
         {
          images[i-1].draw();
            cout << "Exit[0], Image[1-5]: ";
            cin >> i;
         }
}

//    $$ ctor: 1
//    $$ ctor: 2
//    $$ ctor: 3
//    $$ ctor: 4
//    $$ ctor: 5
// Exit[0], Image[1-5]: 2
//    drawing image 2
// Exit[0], Image[1-5]: 4
//    drawing image 4
// Exit[0], Image[1-5]: 2
//    drawing image 2
// Exit[0], Image[1-5]: 0
//    dtor: 5
//    dtor: 4
//    dtor: 3
//    dtor: 2
//    dtor: 1
```

Example 2:

```
// Purpose.  Proxy design pattern

// 1. Design an "extra level of indirection" wrapper class
// 2. The wrapper class holds a pointer to the real class
// 3. The pointer is initialized to null
// 4. When a request comes in, the real object is created "on first use"
//    (aka lazy intialization)
// 5. The request is always delegated

class RealImage
{
   int  id;
public:
   RealImage( int i ) { id = i;  cout << "   $$ ctor: "<< id << '\n'; }
   ~RealImage()              { cout << "   dtor: " << id << '\n'; }
   void draw()               { cout << "   drawing image " << id << '\n'; }
};

// 1. Design an "extra level of indirection" wrapper class
class Image
{
   // 2. The wrapper class holds a pointer to the real class
   RealImage* theRealThing;
   int        id;
   static int next;
public:
   Image()  { id = next++;  theRealThing = 0; }  // 3. Initialized to null
   ~Image() { delete theRealThing; }
   void draw()
         {
            // 4. When a request comes in, the real object is created "on first use"
            if ( ! theRealThing) theRealThing = new RealImage( id );
                  // 5. The request is always delegated
                  theRealThing->draw();
         }
};
int Image::next = 1;

void main( void )
{
   Image images[5];
   int   i;

   cout << "Exit[0], Image[1-5]: ";
   cin >> i;
   while (i)
            {
              images[i-1].draw();
               cout << "Exit[0], Image[1-5]: ";
              cin >> i;
            }
}

// Exit[0], Image[1-5]: 2
//    $$ ctor: 2
//    drawing image 2
// Exit[0], Image[1-5]: 4
//    $$ ctor: 4
//    drawing image 4
// Exit[0], Image[1-5]: 2
//    drawing image 2
// Exit[0], Image[1-5]: 4
//    drawing image 4
// Exit[0], Image[1-5]: 0
//    dtor: 4
//    dtor: 2
```

Example 3:

```
// Purpose.  "->" and "." operators give different results

class Subject { public: virtual void execute() = 0; };

class RealSubject : public Subject
{
   string str;
public:
   RealSubject( string s ) { str = s; }
   /*virtual*/ void execute() { cout << str << '\n'; }
};

class ProxySubject : public Subject
{
   string       first, second, third;
   RealSubject* ptr;
public:
   ProxySubject( string s )
{
      int num = s.find_first_of( ' ' );
      first = s.substr( 0, num );   s = s.substr( num+1 );
      num = s.find_first_of( ' ' );
      second = s.substr( 0, num );  s = s.substr( num+1 );
      num = s.find_first_of( ' ' );
      third = s.substr( 0, num );   s = s.substr( num+1 );
      ptr = new RealSubject( s );
}
   ~ProxySubject() { delete ptr; }
   RealSubject* operator->()
{
    cout << first << ' ' << second << ' ';
    return ptr;
}
  /*virtual*/ void execute()
{
    cout << first << ' ' << third << ' ';
    ptr->execute();
}
};

void main( void )
{
  ProxySubject obj( string( "the quick brown fox jumped over the dog" ) );
  obj->execute();
  obj.execute();
}

// the quick fox jumped over the dog
// the brown fox jumped over the dog
```

Example 4:

```
// Purpose.  Proxy design pattern lab (reference counting)
//
// Discussion.  One of the four common situations in which the Proxy pattern is
// applicable is as a "smart reference" - a replacement for a bare pointer that
// performs additional actions when an object is referenced.  Here, that addi-
// tional action is "reference counting".  The original String class becomes
// the inner "body" class.  A "count" data member is added to the body class.
// A new outer "handle" class is created that contains a pointer to a body
// instance.  Body instances common to multiple handle instances are shared by
// simply incrementing the reference count.  As these shared references go away,
// the reference count is decremented.  When the count goes to zero, the body
// instance is freed.  This mechanism allows the "reference counted class" to
// perform simple-minded garbage collection on itself.
//
// Assignment.
// o Change the class String to an inner body StringRep.
// o Add a count_ private data member to StringRep.  Initialize count_ to 1 in
//   the default ctor and the 1-arg ctor.
// o Create a new outer handle class String. Make String a friend of StringRep.
// o String has one private data member - a pointer to a StringRep (rep_).
// o The String default ctor initializes rep_ with 'new StringRep("")'.
// o The String 1-arg ctor initializes rep_ with 'new StringRep(arg)'.
// o The String copy ctor initializes rep_ with the rep_ of the initializing
//   String.  It also increments the count_ of that rep_.
// o The String dtor decrements the count_ of rep_ and deletes rep_ if count_
//   is 0.
// o The String operator=(): increments the count_ of the "rhs" rep_,
//   decrements the count_ of its own rep_ and deletes rep_ if count_ is 0,
//   assigns the "rhs" rep_ to its own rep_, and returns *this.
// o Add cout statements to String to generate the target output.
// o Make the operator<<() function a friend of both StringRep and String.
//   Modify it as necessary.

#include <iostream.h>
#include <string.h>

class String
{
public:
   friend ostream& operator << ( ostream&, String& );
   String()
   {
      cout << "   String ctor (def):" << endl;
      str_ = NULL;
   }
   String( const char* in )
   {
      cout << "   String ctor: " << in << '.' << endl;
      str_ = new char[strlen(in) + 1];
      strcpy( str_, in );
   }
   String( String& str )
  {
      cout << "   String ctor (copy): " << str.str_ << '.' << endl;
      str_ = new char[strlen(str.str_) + 1];
      strcpy( str_, str.str_ );
   }
   ~String()
  {
      cout << "   String dtor: " << str_ << '.' << endl;
      delete str_;
   }
   String& operator= ( String& rhs )
  {
      if (this == &rhs)  return *this;
      delete str_;
      str_ = new char[strlen(rhs.str_) + 1];
      strcpy( str_, rhs.str_ );
      return *this;
   }
private:
   char*  str_;
};

ostream& operator << ( ostream& os, String& str ) { return os << str.str_; }

void main( void )
{
   String  a( "hello" );
   String  b = "world";
   String  c( a );
   String  d = a;
   String  e;
   a = b;
   e = b;
   cout << "a is " << a << '.' << endl;
  cout << "b is " << b << '.' << endl;
  cout << "c is " << c << '.' << endl;
  cout << "d is " << d << '.' << endl;
  cout << "e is " << e << '.' << endl;
}

/****** Current output ******/
//    String ctor: hello.
//    String ctor: world.
//    String ctor (copy): hello.
//    String ctor (copy): hello.
//    String ctor (def):
// a is world.
// b is world.
// c is hello.
// d is hello.
// e is world.
//    String dtor: world.
//    String dtor: hello.
//    String dtor: hello.
//    String dtor: world.
//    String dtor: world.

/****** Target output ******/
//    StringRep ctor: hello.
// String ctor: hello.
//    StringRep ctor: world.
// String ctor: world.
// String ctor (copy): hello.
// String ctor (copy): hello.
//    StringRep ctor: .
// String ctor (def):
//    StringRep dtor: .
// a is world.
// b is world.
// c is hello.
// d is hello.
// e is world.
// String dtor: world, before decrement, count is 3
// String dtor: hello, before decrement, count is 2
// String dtor: hello, before decrement, count is 1
//    StringRep dtor: hello.
// String dtor: world, before decrement, count is 2
// String dtor: world, before decrement, count is 1
//    StringRep dtor: world.
```

Chain of Responsibility

Intent

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Problem

There is a potentially variable number of "handler" objects and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.

Discussion

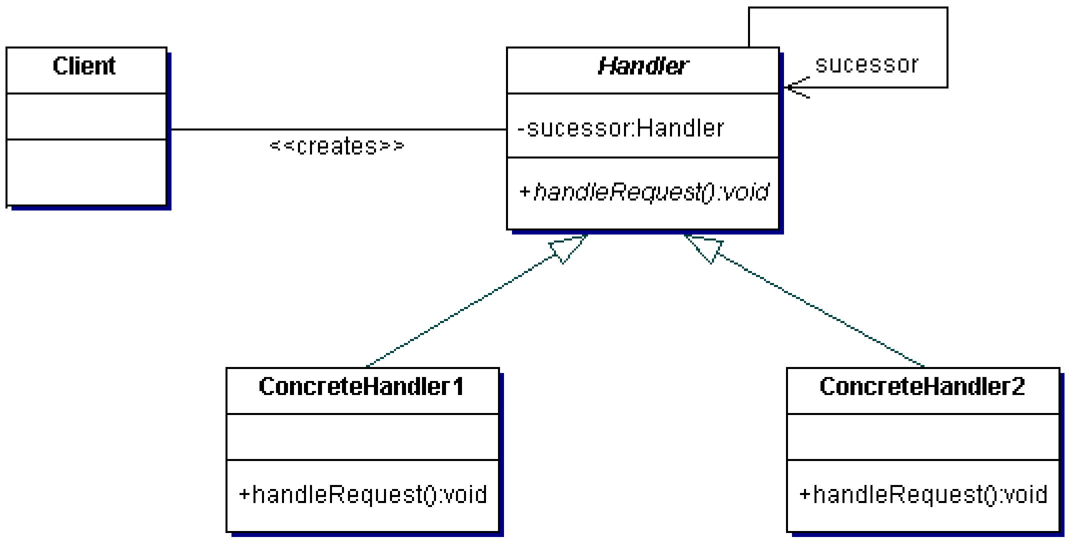
The pattern chains the receiving objects together, and then passes any request messages from object to object until it reaches an object capable of handling the message. The number and type of handler objects isn't known a priori, they can be configured dynamically. The chaining mechanism uses recursive composition to allow an unlimited number of handlers to be linked.

Chain of Responsibility simplifies object interconnections. Instead of senders and receivers maintaining references to all candidate receivers, each sender keeps a single reference to the head of the chain, and each receiver keeps a single reference to its immediate successor in the chain.

Make sure there exists a "safety net" to "catch" any requests which go unhandled.

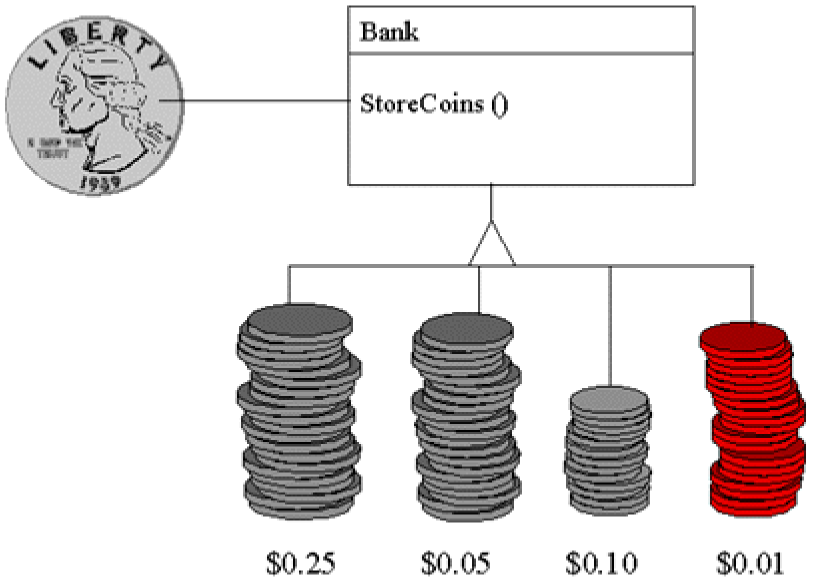
Do not use Chain of Responsibility when each request is only handled by one handler, or, when the client object knows which service object should handle the request.

Structure



Example

The Chain of Responsibility pattern avoids coupling the sender of a request to the receiver by giving more than one object a chance to handle the request. Mechanical coin sorting banks use the Chain of Responsibility. Rather than having a separate slot for each coin denomination coupled with a receptacle for the denomination, a single slot is used. When the coin is dropped, the coin is routed to the appropriate receptacle by the mechanical mechanisms within the bank. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers.

Chain of Responsibility can use Command to represent requests as objects. [GOF, p349]

Chain of Responsibility is often applied in conjunction with Composite. There, a component's parent can act as its successor. [GOF, p232]

Program Examples

Example 1:

```
// Purpose.  Chain of Responsibility.
//
// Discussion.  Instead of the client having to know about the number of
// "handlers", and manually map requests to available handlers;
//design the handlers into an "intelligent" chain.  Clients "launch
// and leave" requests with the head of the chain.

#include <iostream.h>


class H
{
public:
H()
{
id_ = count_++;
busy_ = 0;
}
~H()
{
cout << id_ << " dtor" << endl;
}

int handle()
{
if (busy_ = ! busy_)
        {
                cout << id_ << " handles"<< endl;
                return 1;
        }
 else
        {
                cout << id_ << " is busy"<< endl;
                return 0;
        }
}
private:
        int     id_, busy_;
        static int count_;
};

int H::count_ = 1;

void main( void )
{
const int TOTAL = 2;
H* list[TOTAL] = { new H, new H };

for (int i=0; i < 6; i++)
        for (int j=0; 1 ;j = (j + 1) % TOTAL)
                if (list[j]->handle())
                        break;

for (int k=0; k < TOTAL; k++)
        delete list[k];
}

// 1 handles
// 1 is busy
// 2 handles
// 1 handles
// 1 is busy
// 2 is busy
// 1 handles
// 1 is busy
// 2 handles
// 1 handles
// 1 dtor
// 2 dtor
```

Example 2:

```
#include <iostream.h>

class H
{
public:
H( H* next = 0 )
        {
        id_   = count_++;
        busy_ = 0;
        next_ = next;
        }
~H()
 {
        cout << id_<<" dtor" << endl;
        // don't get into a loop
        if (next_->next_)
                {
                H* t = next_;
                next_ = 0;
                delete t;
                }
}
void setNext( H* next )
{
next_ = next;
}
void handle()
{
if (busy_ = ! busy_)
        cout << id_ << " handles"<< endl;
else
        {
        cout << id_ << " is busy"<< endl;
        next_->handle();
        }
}
private:
        int     id_, busy_;
        H*       next_;
        static int count_;
};

int H::count_ = 1;

H* setUpList()
{
H* first = new H;
H* last = new H(first);
first->setNext( last );
return first;
}

void main( void )
{
H* head = setUpList();

for (int i=0; i < 6; i++)
        head->handle();

delete head;
}

// 1 handles
// 1 is busy
// 2 handles
// 1 handles
// 1 is busy
// 2 is busy
// 1 handles
// 1 is busy
// 2 handles
// 1 handles
// 1 dtor
// 2 dtor
```

Example 3:

```
// Purpose.  Chain of Responsibility design pattern

// 1. Put a "next" pointer in the base class
// 2. The "chain" method in the base class always delegates to the next object
// 3. If the derived classes cannot handle, they delegate to the base class

#include <iostream>
#include <vector>
#include <ctime>
using namespace std;

class Base
 {
   Base* next;                // 1. "next" pointer in the base class
public:
   Base() { next = 0; }
   void setNext( Base* n ) { next = n; }
   void add( Base* n ) { if (next) next->add( n ); else next = n; }
   // 2. The "chain" method in the base class always delegates to the next obj
   virtual void handle( int i ) { next->handle( i ); }
};

class Handler1 : public Base
{
public:
   /*virtual*/ void handle( int i )
         {
          if (rand() % 3)
                  {      // 3. Don't handle requests 3 times out of 4
                        cout << "H1 passsed " << i << "  ";
                        Base::handle( i );   // 3. Delegate to the base class
                  }
         else
                  cout << "H1 handled " << i << "  ";
         }
};

class Handler2 : public Base
{
 public:
   /*virtual*/ void handle( int i )
         {
               if (rand() % 3)
                   {
                         cout << "H2 passsed " << i << "  "; Base::handle( i );
                  }
               else cout << "H2 handled " << i << "  ";
         }
};


class Handler3 : public Base
{
public:
   /*virtual*/ void handle( int i )
          {
                    if (rand() % 3)
                             {
                                 cout << "H3 passsed " << i << "  "; Base::handle( i );
                             }
                    else
                                 cout << "H3 handled " << i << "  ";
          }
};

void main( void )
{
   srand( time( 0 ) );
   Handler1 root;   Handler2 two;   Handler3 thr;
   root.add( &two );
   root.add( &thr );
   thr.setNext( &root );
   for (int i=1; i < 10; i++)
          {
             root.handle( i );
             cout << '\n';
          }
}

// H1 passsed 1  H2 passsed 1  H3 passsed 1  H1 passsed 1  H2 handled 1
// H1 handled 2
// H1 handled 3
// H1 passsed 4  H2 passsed 4  H3 handled 4
// H1 passsed 5  H2 handled 5
// H1 passsed 6  H2 passsed 6  H3 passsed 6  H1 handled 6
// H1 passsed 7  H2 passsed 7  H3 passsed 7  H1 passsed 7  H2 handled 7
// H1 handled 8
// H1 passsed 9  H2 passsed 9  H3 handled 9
```

Example 4:

```
// Purpose.  Chain of Responsibility and Composite

// 1. Put a "next" pointer in the base class
// 2. The "chain" method in the base class always delegates to the next object
// 3. If the derived classes cannot handle, they delegate to the base class

#include <iostream>
#include <vector>
#include <ctime>
using namespace std;

class Component
 {
   int        value;
   Component* next;             // 1. "next" pointer in the base class

public:
   Component( int v, Component* n ) { value = v;  next = n; }
   void setNext( Component* n )     { next = n; }
```

```
  virtual void traverse()         { cout << value << ' '; }
  // 2. The "chain" method in the base class always delegates to the next obj
  virtual void volunteer()        { next->volunteer(); }
};

class Primitive : public Component
 {
public:
  Primitive( int val, Component* n = 0 ) : Component( val, n ) { }
  /*virtual*/ void volunteer()
    {
    Component::traverse();
    // 3. Primitive objects don't handle volunteer requests 5 times out of 6
    if (rand() * 100 % 6 != 0)
          // 3. Delegate to the base class
              Component::volunteer();
    }
};

class Composite : public Component
{
  vector<Component*> children;
public:
  Composite( int val, Component* n = 0 ) : Component( val, n ) { }
  void add( Component* c ) { children.push_back( c ); }
  /*virtual*/ void traverse()
   {
    Component::traverse();
    for (int i=0; i < children.size(); i++)
           children[i]->traverse();
   }
  // 3. Composite objects never handle volunteer requests
  /*virtual*/ void volunteer() { Component::volunteer(); }
};

void main( void )
{
  srand( time( 0 ) );                          // 1
  Primitive seven( 7 );                        // |
  Primitive six( 6, &seven );                  // +-- 2
  Composite three( 3, &six );                  // |   |
  three.add( &six );  three.add( &seven );     // |   +-- 4 5
  Primitive five( 5, &three );                 // |
  Primitive four( 4, &five );                  // +-- 3
  Composite two( 2, &four );                   // |   |
  two.add( &four );   two.add( &five );        // |   +-- 6 7
  Composite one( 1, &two );                    // |
  Primitive nine( 9, &one );                   // +-- 8 9
  Primitive eight( 8, &nine );
  one.add( &two );  one.add( &three );  one.add( &eight );  one.add( &nine );
  seven.setNext( &eight );
  cout << "traverse: ";   one.traverse();   cout << '\n';
  for (int i=0; i < 8; i++)
        {
             one.volunteer();  cout << '\n';
            }
}

// traverse: 1 2 4 5 3 6 7 8 9
// 4
// 4 5 6 7
// 4 5 6 7 8 9 4 5 6 7 8 9 4
// 4
// 4 5 6
// 4 5
// 4 5
// 4 5 6 7 8 9 4 5 6 7 8 9 4 5 6
```

Example 5:

```
// Purpose.  Chain of Responsibility - links bid on job, chain assigns job
//
// 1. Base class maintains a "next" pointer
// 2. Each "link" does its part to handle (and/or pass on) the request
// 3. Client "launches and leaves" each request with the chain
// 4. The current bid and bidder are maintained in chain static members
// 5. The last link in the chain assigns the job to the low bidder

#include <iostream>
#include <ctime>
using namespace std;

class Link
{
   int   id;
   Link* next;                       // 1. "next" pointer
   static int   theBid;              // 4. Current bid and bidder
   static Link* bidder;
public:
   Link( int num ) { id = num;  next = 0; }
   void addLast( Link* l )
   {
      if (next) next->addLast( l );  // 2. Handle and/or pass on
      else      next = l;
   }
   void bid()
     {
      int num = rand() % 9;
      cout << id << '-' << num << "  ";
      if (num < theBid)
            {
                  theBid = num;              // 4. Current bid and bidder
                  bidder = this;
            }


      if (next)
            next->bid();          // 2. Handle and/or pass on
      else
```

```
            bidder->execute();   // 5. The last 1 assigns the job
   }

void execute()
  {
     cout << id << " is executing\n";
     theBid = 999;
  }
};

int   Link::theBid = 999;            // 4. Current bid and bidder
Link* Link::bidder = 0;

void main( void )
{
   Link chain( 1 );
   for (int i=2; i < 7; i++)
      chain.addLast( new Link( i ) );
   srand( time( 0 ) );
   for (i=0; i < 10; i++)
      chain.bid();                  // 3. Client "launches & leaves"
}

// 1-1  2-6  3-0  4-3  5-1  6-0  3 is executing
// 1-1  2-1  3-1  4-0  5-7  6-1  4 is executing
// 1-0  2-1  3-0  4-6  5-1  6-2  1 is executing
// 1-6  2-3  3-8  4-0  5-1  6-4  4 is executing
// 1-8  2-0  3-5  4-8  5-4  6-2  2 is executing
// 1-1  2-0  3-8  4-8  5-7  6-0  2 is executing
// 1-1  2-1  3-3  4-1  5-6  6-2  1 is executing
// 1-2  2-1  3-0  4-3  5-6  6-8  3 is executing
// 1-7  2-5  3-4  4-2  5-1  6-2  5 is executing
// 1-3  2-6  3-7  4-7  5-6  6-0  6 is executing
```

Example 6:

```
// Purpose.  Chain of Responsibility design pattern lab
//
// Problem.  Four Slot objects are being used, and the client has to know
// about and interact with all of them.  It would be preferable to decouple
// the client from its many servers by encapsulating all the Slot objects
// in a single pipeline, or chain, abstraction.  The advantages are:
// "request handling" algorithms are hidden in the pipeline abstraction, and
// client complexity is reduced because it can now "launch and leave" all
// requests with the head of the pipeline and the pipeline "divides and
// conquers" the request across its collection of processing elements.
//
// Assignment.
// o Add a Slot* data member to Slot that will point to each Slot's successor
//   in the chain
// o Modify the constructors for Slot, Quarter, Dime, Nickel, and Penny to
//   accept a Slot* and initialize the Slot* data member (use an "=0" default
//   argument)
// o Change accept() to delegate to the successor if the conditional expression
//   does not succeed (accept() can now have a void return type)
// o Add a report() member function to the Slots hierarchy that encapsulates
//   the functionality of the final four cout's in main().
// o Encapsulate the slots array set-up in a free function.  Create the "chain"
//   in this function and return it to main().
// o Change main() to eliminate all knowledge of, or interaction with, the
//   number and type of Slot objects.

#include <iostream.h>
#include <stdlib.h>
#include <time.h>

class Slot
{
public:
  Slot( int val )
        {
             value_ = val;
             count_ = 0;
        }
  int accept( int coin )
{
    if (coin == value_)
        {
               count_++;
               return 1;
        }
    return 0;
}

int getCount()
{
    return count_;
}
private:
  int  value_;
  int  count_;
};

class Quarter : public Slot
{
public:
  Quarter() : Slot( 25 ) { }
};

class Dime : public Slot
{
public:
  Dime() : Slot( 10 ) { }
};

class Nickel : public Slot
{
public:
  Nickel() : Slot( 5 ) { }
};
```

```
class Penny : public Slot
{
public:
  Penny() : Slot( 1 ) { }
};

int pickCoin()
{
  static int choices[4] = { 1, 5, 10, 25 };
  return choices[rand() % 4];
}

void main( void )
{
  Slot*    slots[4];
  slots[0] = new Quarter;
  slots[1] = new Dime;
  slots[2] = new Nickel;
  slots[3] = new Penny;
  int      i, coin, total;
  time_t   t;
  srand((unsigned) time(&t));

  for (i=0, total=0; i < 10; i++)
  {
    coin = pickCoin();
    cout << coin << " ";
    total += coin;
    for (int j=0; j < 4; j++)
      if (slots[j]->accept( coin )) break;
  }

  cout << "\ntotal deposited is $" << total/100 << "."
    << (total%100 < 10 ? "0" : "") << total%100 << endl;
  cout << "quarters - " << slots[0]->getCount() << endl;
  cout << "dimes    - " << slots[1]->getCount() << endl;
  cout << "nickels  - " << slots[2]->getCount() << endl;
  cout << "pennies  - " << slots[3]->getCount() << endl;
}

// 10 5 25 5 1 25 5 25 1 25
// total deposited is $1.27
// quarters - 4
// dimes    - 1
// nickels  - 3
// pennies  - 2
//
// 25 25 10 5 1 5 10 5 1 5
// total deposited is $0.92
// quarters - 2
// dimes    - 2
// nickels  - 4
// pennies  - 2
//
```

```
// 5 10 25 10 25 1 25 10 1 25
// total deposited is $1.37
// quarters - 4
// dimes    - 3
// nickels  - 1
// pennies  - 2
```

Command

Intent

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Problem

Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

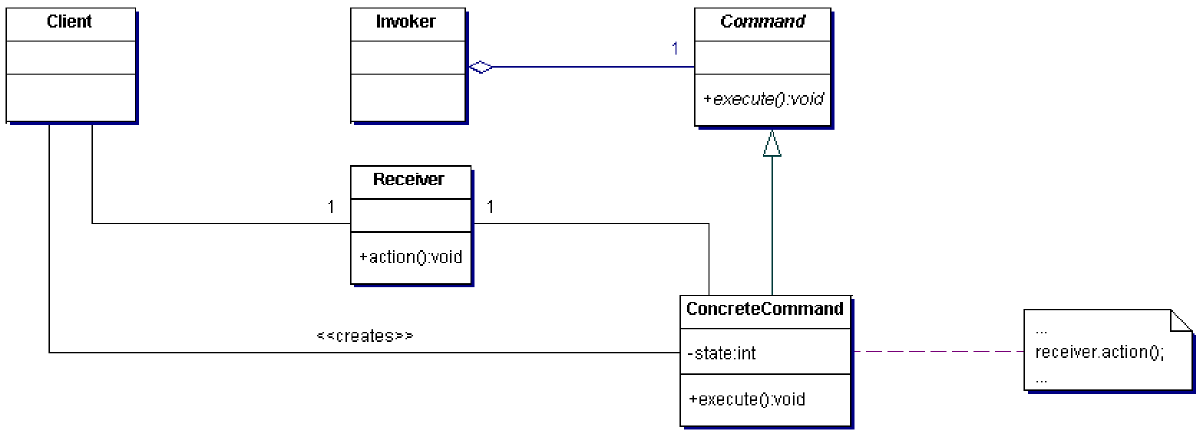
Discussion

Command decouples the object that invokes the operation from the one that knows how to perform it. To achieve this separation, the designer creates an abstract base class that maps a receiver (an object) with an action (a pointer to a member function). The base class contains an execute() method that simply calls the action on the receiver.

All clients of Command objects treat each object as a "black box" by simply invoking the object's virtual execute() method whenever the client requires the object's "service".

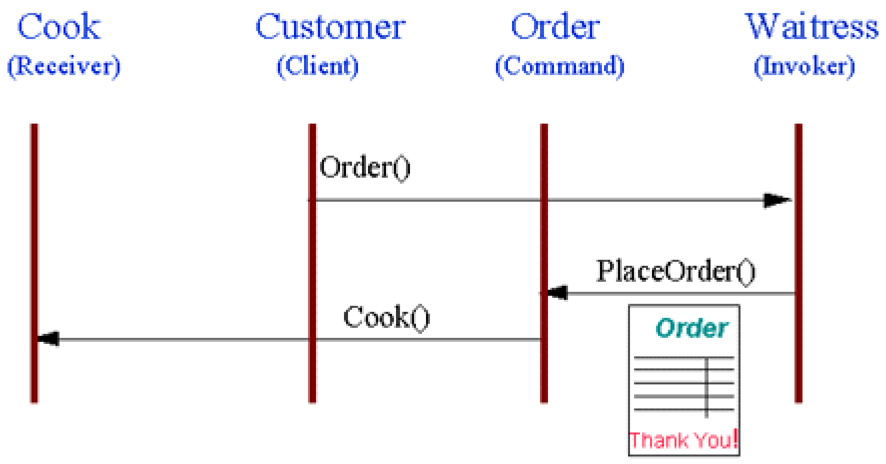
Sequences of Command objects can be assembled into composite (or macro) commands.

Structure



Example

The Command pattern allows requests to be encapsulated as objects, thereby allowing clients to be parameterized with different requests. The "check" at a diner is an example of a Command pattern. The waiter or waitress takes an order or command from a customer and encapsulates that order by writing it on the check. The order is then queued for a short order cook. Note that the pad of "checks" used by each waiter is not dependent on the menu, and therefore they can support commands to cook many different items. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Command normally specifies a sender-receiver connection with a subclass.

Chain of Responsibility can use Command to represent requests as objects. [GOF, p349].

Command and Memento act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time. Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value. [GOF, p346]

Command can use Memento to maintain the state required for an undo operation. [GOF, p242]

MacroCommands can be implemented with Composite. [GOF, p242]

A Command that must be copied before being placed on a history list acts as a Prototype. [GOF, p242]

POSA's Command Processor pattern describes the scaffolding Command needs to support full multilevel undo and redo. [Vlissides, *Java Report*, Nov 2000, p80]

Notes

"Memento-Command" - a new pattern by John Vlissides. Presented in *Java Report*, November 2000, pp70-80. Intent: manage undo state when a command can affect unforseen receivers.

Program Examples

Example 1:

```
// Purpose.  Command
//
// Discussion.  On the left, an IOU has been encapsulated as a struct,
// and TheBoss class is tightly coup led to that struct.  On the right,
// TheBoss is only coupled to the abstract class Command.  Lots of pos-
// sible derived classes could be substituted: IOUs that call payUp() on
// Deadbeats, Checks that call cash() on Banks, Stocks that call redeem()
// on Companies.  Each "command" is a "token" that gets transfered from
// one holder to another, until some one chooses to "execute" it.

class Deadbeat
{
 public:
Deadbeat( int v ) { cash_ = v; }
int payUp( int v ) {cash_ -= v;  return v; }
int rptCash() { return cash_; }
private:
        int cash_;
};

struct IOU
{
Deadbeat*  objPtr;
int (Deadbeat::*funPtr)( int );
int     amt;
};
```

```
class Enforcer
{
 public:
Enforcer( IOU& m ) : mkr_(m) { }
IOU& collect() { return mkr_; }
private:
        IOU& mkr_;
};

class TheBoss
{
public:
TheBoss() { cash_ = 1000; }
collect( IOU& i )
{
cash_ +=((i.objPtr)->*i.funPtr)(i.amt)
}

int rptCash() { return cash_; }
private:
        int cash_;
};

void main( void )
{
Deadbeat joe(90), tom(90);
IOU one ={&joe, &Deadbeat::payUp,60}
IOU two ={&tom, &Deadbeat::payUp,70}
Enforcer quido(one), lucca(two);
TheBoss  don;

don.collect( quido.collect() );
don.collect( lucca.collect() );
cout << "joe has $" << joe.rptCash()
cout << "tom has $" << tom.rptCash()
cout << "don has $" << don.rptCash()
}
```

Example 2:

```
class Deadbeat
{
public:
Deadbeat( int v ) { cash_ = v; }
int payUp( int v ) {cash_ -= v;  return v; }
int rptCash() { return cash_; }
private:
        int cash_;
};
class Command
{
 public:
        virtual int execute() = 0;
};
```

```
class IOU : public Command
{
public:
typedef int (Deadbeat::*Meth)(int);
IOU( Deadbeat* r, Meth a, int m )
{
        obj_ = r;
        mth_ = a;
        amt_ = m;
}
int execute()
{
return (obj_->*mth_)( amt_ );
}
private:
        Deadbeat*  obj_;
        Meth       mth_;
        int        amt_;
};

class Enforcer
{
public:
Enforcer( Command& c ) : cmd_(c) {}
Command& collect() { return cmd_; }
private:
        Command& cmd_;
};

class TheBoss
{
public:
TheBoss() { cash_ = 1000; }
collect( Command& cmd ) {cash_ += cmd.execute(); }
int rptCash() { return cash_; }
private:
        int cash_;
};

void main( void )
{
Deadbeat joe(90), tom(90);
IOU one(&joe, &Deadbeat::payUp, 60);
IOU two(&tom, &Deadbeat::payUp, 70);
Enforcer quido(one), lucca(two);
TheBoss  don;

don.collect( quido.collect() );
don.collect( lucca.collect() );
cout << "joe has $" << joe.rptCash()
cout << "tom has $" << tom.rptCash()
cout << "don has $" << don.rptCash()
}
```

```
// joe has $30
// tom has $20
// don has $1130
```

Example 3:

```
// Purpose.  Command design pattern

// 1. Create a class that encapsulates some number of the following:
//       a "receiver" object
//       the method to invoke
//       the arguments to pass
// 2. Instantiate an object for each "callback"
// 3. Pass each object to its future "sender"
// 4. When the sender is ready to callback to the receiver, it calls execute()

#include <iostream>
#include <string>
using namespace std;
class Person;

class Command
{
                                   // 1. Create a class that encapsulates an object
   Person*  object;                //    and a member function
   void (Person::*method)();      // a pointer to a member function (the attri-
public:                           //    bute's name is "method")
   Command( Person* obj = 0, void (Person::*meth)() = 0 )
   {
      object = obj;                // the argument's name is "meth"
      method = meth;
   }
   void execute()
   {
      (object->*method)();     // invoke the method on the object
    }
};

class Person
{
   string   name;
   Command  cmd;                  // cmd is a "black box", it is a method invoca-
public:                           //    tion promoted to "full object status"
   Person( string n, Command c ) : cmd( c )
   {
      name = n;
   }
   void talk()
   {                               // "this" is the sender, cmd has the receiver
      cout << name << " is talking" << endl;
      cmd.execute();               // ask the "black box" to callback the receiver
   }
```

```
 void passOn()
  {
     cout << name << " is passing on" << endl;
     cmd.execute();          // 4. When the sender is ready to callback to
   }                        //    the receiver, it calls execute()
   void gossip()
  {
     cout << name << " is gossiping" << endl;
     cmd.execute();
   }
   void listen()
  {
     cout << name << " is listening" << endl;
   }
};

void main( void )
{
   // Fred will "execute" Barney which will result in a call to passOn()
   // Barney will "execute" Betty which will result in a call to gossip()
   // Betty will "execute" Wilma which will result in a call to listen()
   Person wilma(  "Wilma",  Command() );
   // 2. Instantiate an object for each "callback"
   // 3. Pass each object to its future "sender"
   Person betty(  "Betty",  Command( &wilma,  &Person::listen ) );
   Person barney( "Barney", Command( &betty,  &Person::gossip ) );
   Person fred(   "Fred",   Command( &barney, &Person::passOn ) );
   fred.talk();
}

// Fred is talking
// Barney is passing on
// Betty is gossiping
// Wilma is listening
```

Example 4:

```
// Purpose.  Simple and "macro" commands
// Discussion.  Encapsulate a request as an object.  SimpleCommand
// maintains a binding between a receiver object and an action stored as a
// pointer to a member function.  MacroCommand maintains a sequence of
// Commands.  No explicit receiver is required because the subcommands
// already define their receiver.  MacroCommand may contain MacroCommands.

#include <iostream>
#include <vector>
using namespace std;

class Number
{
 public:
   void dubble( int& value ) { value *= 2; }
};
```

```
class Command { public: virtual void execute( int& ) = 0; };

class SimpleCommand : public Command
{
  typedef void (Number::* Action)(int&);
  Number* receiver;
  Action  action;
public:
  SimpleCommand( Number* rec, Action act )
{
    receiver = rec;
    action = act;
}
  /*virtual*/ void execute( int& num ) { (receiver->*action)( num ); }
};

class MacroCommand : public Command
{
  vector<Command*> list;
public:
  void add( Command* cmd ) { list.push_back( cmd ); }
  /*virtual*/ void execute( int& num )
        {
        for (int i=0; i < list.size(); i++)
              list[i]->execute( num );
        }
};

void main( void )
{
  Number object;
  Command* commands[3];
  commands[0] = &SimpleCommand( &object, &Number::dubble );

  MacroCommand two;
  two.add( commands[0] );   two.add( commands[0] );
  commands[1] = &two;

  MacroCommand four;
  four.add( &two );   four.add( &two );
  commands[2] = &four;

  int num, index;
  while (true)
        {
            cout << "Enter number selection (0=2x 1=4x 2=16x): ";
            cin >> num >> index;
            commands[index]->execute( num );
            cout << "   " << num << '\n';
        }
}
```

```
// Enter number selection (0=2x 1=4x 2=16x): 3 0
//    6
// Enter number selection (0=2x 1=4x 2=16x): 3 1
//    12
// Enter number selection (0=2x 1=4x 2=16x): 3 2
//    48
// Enter number selection (0=2x 1=4x 2=16x): 4 0
//    8
// Enter number selection (0=2x 1=4x 2=16x): 4 1
//    16
// Enter number selection (0=2x 1=4x 2=16x): 4 2
//    64
```

Example 5:

```
// Purpose.  Command design pattern and inheritance
//
// Discussion.  The Command pattern promotes a deferred method invocation to
// full object status.  Each Command object is a "black box" - it is opaque to
// its holder/user.  Here, a portfolio's heterogeneous collection of financial
// instruments is being treated homogeneously, because, they all inherit from
// a common base class, and, they all have a "convert to currency" method with
// a common signature.

#include <iostream>
using namespace std;

class Instrument { public: virtual ~Instrument() { } };

class IOU : public Instrument
{
   int amount;
public:
   IOU( int in ) { amount = in; }
   int payUp() { return amount; }
};

class Check : public Instrument
{
   int amount;
public:
   Check( int in ) { amount = in; }
   int cash() { return amount; }
};

class Stock : public Instrument
{
   int amount;
public:
   Stock( int in ) { amount = in; }
   int redeem() { return amount; }
};
```

```
class Command
{
public:
   typedef int (Instrument::*Action)();
   Command( Instrument* o, Action m )
   {
      object = o;
      method = m;
   }
   int execute() { return (object->*method)(); }
private:
   Instrument* object;
   Action      method;
};

void main( void )
{
   Command* portfolio[] = {  // old C cast, or new RTTI is required
      &Command( &IOU(100),   (int(Instrument::*)())&IOU::payUp ),
      &Command( &Check(200), static_cast<int(Instrument::*)()>(&Check::cash) ),
      &Command( &Stock(300), static_cast<int(Instrument::*)()>(&Stock::redeem) ) };

   for (int netWorth=0, i=0; i < 3; i++)
            netWorth += portfolio[i]->execute();
   cout << "net worth is now " << netWorth << '\n';
}

// net worth is now 600
```

Example 6:

```
// Purpose.  Command design pattern and class template

#include <iostream>
#include <string>
using namespace std;

class A
{
   int divisor;
public:
   A( int div ) { divisor = div; }
   int divide( int in ) { return in / divisor; }
   int modulus( int in ) { return in % divisor; }
};

class B
{
   string str;
public:
   B( string s ) { str = s; }
   string prepend(  string in ) { return in + str; }
```

```
  string postpend( string in ) { return str + in; }
};

template <typename CLS, typename ARG>class Command
{
public:
  typedef ARG (CLS::*Action)( ARG );
  Command( CLS* o, Action m, ARG a )
   {
     object = o;
     method = m;
     argument = a;
   }
  ARG execute() { return (object->*method)( argument ); }
private:
  CLS*   object;
  Action method;
  ARG    argument;
};

void main( void )
{
  Command<A,int>* list1[4] = {
    &Command<A,int>( &A(3), &A::divide,  16 ),
    &Command<A,int>( &A(3), &A::modulus, 16 ),
    &Command<A,int>( &A(4), &A::divide,  16 ),
    &Command<A,int>( &A(4), &A::modulus, 16 ) };

  for (int i=0; i < 4; i++)
    cout << "numbers are " << list1[i]->execute() << '\n';

  B obj("abc");
  Command<B,string>* list2[4] = {
    new Command<B,string>( &obj, &B::prepend,  "123" ),
    new Command<B,string>( &obj, &B::prepend,  "xyz" ),
    new Command<B,string>( &obj, &B::postpend, "123" ),
    new Command<B,string>( &obj, &B::postpend, "xyz" ) };

  for (i=0; i < 4; i++)
    cout << "strings are " << list2[i]->execute() << '\n';
}

// numbers are 5
// numbers are 1
// numbers are 4
// numbers are 0
// strings are 123abc
// strings are xyzabc
// strings are abc123
// strings are abcxyz
```

Example 7:

```
// Purpose.  Command/Adapter design pattern (External Polymorphism demo)

// 1. Specify the new desired interface
// 2. Design a "wrapper" class that can "impedance match" the old to the new
// 3. The client uses (is coupled to) the new interface
// 4. The wrapper/adapter "maps" to the legacy implementation

#include <iostream.h>
class ExecuteInterface
{
public:                                      // 1. Specify the new i/f
   virtual ~ExecuteInterface() { }
   virtual void execute() = 0;
};
// 2. Design a "wrapper" or  "adapter" class
template <class TYPE>  class ExecuteAdapter : public ExecuteInterface
{
public:
   ExecuteAdapter( TYPE* o, void (TYPE::*m)() ) { object = o;  method =m; }
   ~ExecuteAdapter()                         { delete object; }
   // 4. The adapter/wrapper "maps" the new to the legacy implementation
   void execute()            /* the new */
         {
                  (object->*method)();
         }
private:
   TYPE* object;                                      // ptr-to-object attribute
   void (TYPE::*method)();              /* the old */    // ptr-to-member-function  attribute

// The old: three totally incompatible classes    // no common base class,
class Fea
{
public:                              // no hope of polymorphism
   ~Fea()       { cout << "Fea::dtor" << endl; }
   void doThis() { cout << "Fea::doThis()" << endl; }
};

class Feye
{
public:
   ~Feye()      { cout << "Feye::dtor" << endl; }
   void doThat() { cout << "Feye::doThat()" << endl; }
};

class Pheau
{
public:
   ~Pheau()         { cout << "Pheau::dtor" << endl; }
   void doTheOther() { cout << "Pheau::doTheOther()" << endl; }
};
```

```
/* the new is returned */ ExecuteInterface** initialize()
 {
   ExecuteInterface** array = new ExecuteInterface*[3];                /* the old is below */
   array[0] = new ExecuteAdapter<Fea>(   new Fea(),     &Fea::doThis);
   array[1] = new ExecuteAdapter<Feye>(  new Feye(),    &Feye::doThat);
   array[2] = new ExecuteAdapter<Pheau>( new Pheau(),   &Pheau::doTheOther );
   return array;
}

void main( void )
 {
   ExecuteInterface** objects = initialize();

   for (int i=0; i < 3; i++) objects[i]->execute();  // 3. Client uses the new
(polymporphism)
   for (i=0; i < 3; i++) delete objects[i];
   delete objects;
}

// Fea::doThis()
// Feye::doThat()
// Pheau::doTheOther()
// Fea::dtor
// Feye::dtor
// Pheau::dtor
```

Example 8:

```
// Purpose.  Command design pattern demo
//
// Discussion.  Encapsulate a request as an object.  SimpleCommand
// maintains a binding between a receiver object and an action stored as a
// pointer to a member function.  MacroCommand maintains a sequence of
// Commands.  No explicit receiver is required because the subcommands
// already define their receiver.  MacroCommand may contain MacroCommands.

#include <iostream.h>
class Number
{
public:
        Number( int value ) { _value = _copy = value; }
        int  getValue()     { return _value; }
        void increment()    { _copy = _value++; }
        void decrement()    { _copy = _value--; }
        void restore()      { _value = _copy; }
        void dubble()       { _copy = _value;  _value = 2 * _value; }
        void half()         { _copy = _value;  _value = _value / 2; }
        void square()       { _copy = _value;  _value = _value * _value; }
private:
        int _value;
        int _copy;
};
```

```
class Command
{
public:
        virtual void execute() = 0;
        virtual void add( Command* ) { }
protected:
        Command() { }
};

class SimpleCommand : public Command
{
public:
        typedef void (Number::* Action)();
        SimpleCommand( Number* receiver, Action action )
         {
                _receiver = receiver;
                _action = action;
        }
        virtual void execute() { (_receiver->*_action)(); }
protected:
        Number* _receiver;
        Action  _action;
};

class MacroCommand : public Command
{
public:
        MacroCommand() { _numCommands = 0; }
        void add( Command* cmd ) { _list[_numCommands++] = cmd; }
        virtual void execute();
private:
        Command* _list[10];
        int      _numCommands;
};

void MacroCommand::execute()
{
        for (int i=0; i < _numCommands; i++)
                _list[i]->execute();
}

void main()
{
        int i;
        cout << "Integer: ";
        cin >> i;
        Number*   object = new Number(i);

        Command*  commands[9];
        commands[1] = new SimpleCommand( object, &Number::increment );
        commands[2] = new SimpleCommand( object, &Number::decrement );
        commands[3] = new SimpleCommand( object, &Number::dubble );
        commands[4] = new SimpleCommand( object, &Number::half );
        commands[5] = new SimpleCommand( object, &Number::square );
        commands[6] = new SimpleCommand( object, &Number::restore );
        commands[7] = new MacroCommand;
        commands[7]->add( commands[1] );
        commands[7]->add( commands[3] );
        commands[7]->add( commands[5] );
        commands[8] = new MacroCommand;
        commands[8]->add( commands[7] );
        commands[8]->add( commands[7] );

cout << "Exit[0], ++[1], --[2], x2[3], Half[4], Square[5], "<< "Undo[6], do3[7] do6[8]: ";
        cin >> i;

        while (i)
        {
                commands[i]->execute();
                cout << "   " << object->getValue() << endl;
cout << "Exit[0], ++[1], --[2], x2[3], Half[4], Square[5], "<< "Undo[6], do3[7] do6[8]: ";
                cin >> i;
        }
}

// Integer: 4
// Exit[0], ++[1], --[2], x2[3], Half[4], Square[5], Undo[6], do3[7] do6[8]: 1
//    5
// Exit[0], ++[1], --[2], x2[3], Half[4], Square[5], Undo[6], do3[7] do6[8]: 3
//    10
// Exit[0], ++[1], --[2], x2[3], Half[4], Square[5], Undo[6], do3[7] do6[8]: 2
//    9
// Exit[0], ++[1], --[2], x2[3], Half[4], Square[5], Undo[6], do3[7] do6[8]: 4
//    4
// Exit[0], ++[1], --[2], x2[3], Half[4], Square[5], Undo[6], do3[7] do6[8]: 5
//    16
// Exit[0], ++[1], --[2], x2[3], Half[4], Square[5], Undo[6], do3[7] do6[8]: 6
//    4
// Exit[0], ++[1], --[2], x2[3], Half[4], Square[5], Undo[6], do3[7] do6[8]: 6
//    4
// Exit[0], ++[1], --[2], x2[3], Half[4], Square[5], Undo[6], do3[7] do6[8]: 7
//    100
// Exit[0], ++[1], --[2], x2[3], Half[4], Square[5], Undo[6], do3[7] do6[8]: 6
//    10
// Exit[0], ++[1], --[2], x2[3], Half[4], Square[5], Undo[6], do3[7] do6[8]: 8
//    940900
```

Example 9:

```
// Purpose.  Command design pattern lab
//
// Problem.  Encapsulation is an excellent tool for mitigating complexity.
// Here, the notion of a "command" has been designed that encapsulates a
// request to be performed on a "second party" object.  This encapsulating
// abstraction can now be handed to a "third party" object that provides
// some interesting "service" for the "first party" application.  As far as
// the Queue object is concerned, Command objects are opaque black boxes.
// This demo models the "first party" as really two separate entities: the
// entity responsible for "queueing up" Commands, and the entity responsible
// for "dequeueing" and executing commands.  The latter entity is obliged to
// "query" the type of each Command as it comes off the queue, and then step
// through a multiple branch conditional construct to identify the correct
// action to take.  Optimally, none of this would be necessary if the
// Command abstraction were appropriately "objectified".
//
// Assignment.
//  Convert Command from a struct to a class
//  Remove the "type" field from the Command class
//  Add the following "pointer to member function" declaration to Command
//  "typedef void (File::* Action )();"
//  Add an "Action" private data member to Command
//  Replace the "unarchive", "transfer", and "compress" initialization
//  arguments with arguments of the form "&File::unarchive"
//  Add an execute() method to Command that simply calls action_ on receiver_
//  ( "(receiver_->*action_)()" )
//  Remove the "type query" expressions from main()
//  Replace the multiple branch conditional construct with a single call to
//  execute()

#include <iostream.h>
#include <string.h>

struct Command;

class Queue
{
public:
   Queue() { add_ = remove_ = 0; }
   void enque( Command* c )
            {
                  array_[add_] = c;
                  add_ = (add_ + 1) % SIZE;
            }
               Command* deque()
            {
                  int temp = remove_;
                  remove_ = (remove_ + 1) % SIZE;
                  return array_[temp];
            }
private:
   enum { SIZE = 10 };
   Command*  array_[SIZE];
   int       add_;
   int       remove_;
};

class File
{
public:
   File( char* n ) { strcpy( name_, n ); }
   void unarchive() { cout << "unarchive " << name_ << endl; }
   void compress()  { cout << "compress "  << name_ << endl; }
   void transfer()  { cout << "transfer "  << name_ << endl; }
private:
   char  name_[30];
};
```

```
enum Action { unarchive, transfer, compress };

struct Command
{
  Command( File* f, Action a ) { receiver = f; action = a; }
  File*   receiver;
  Action  action;
};

Command* input[8] = {
  new Command( new File("irImage.dat"),    unarchive ),
  new Command( new File("screenDump.jpg"), transfer ),
  new Command( new File("paper.ps"),       unarchive ),
  new Command( new File("widget.tar"),     compress ),
  new Command( new File("esmSignal.dat"),  unarchive ),
  new Command( new File("msword.exe"),     transfer ),
  new Command( new File("ecmSignal.dat"),  compress ),
  new Command( new File("image.gif"),      transfer )
};

void main( void )
{
  Queue     que;
  Command*  cmd;
  int       i;

  for (i=0; i < 8; i++)
    que.enque( input[i] );

  for (i=0; i < 8; i++)
  {
    cmd = que.deque();
    if (cmd->action == unarchive)
              cmd->receiver->unarchive();
    else if (cmd->action == transfer)
              cmd->receiver->transfer();
    else if (cmd->action == compress)
              cmd->receiver->compress();
  }
}

// unarchive irImage.dat
// transfer screenDump.jpg
// unarchive paper.ps
// compress widget.tar
// unarchive esmSignal.dat
// transfer msword.exe
// compress ecmSignal.dat
// transfer image.gif
```

Interpreter

Intent

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

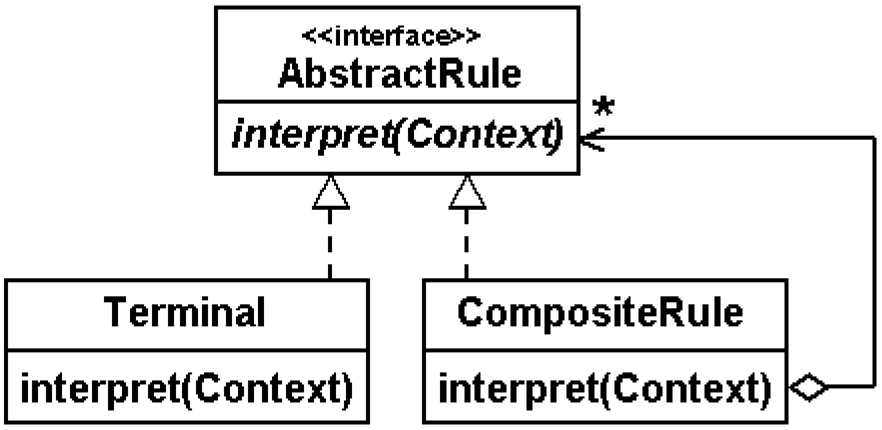
Problem

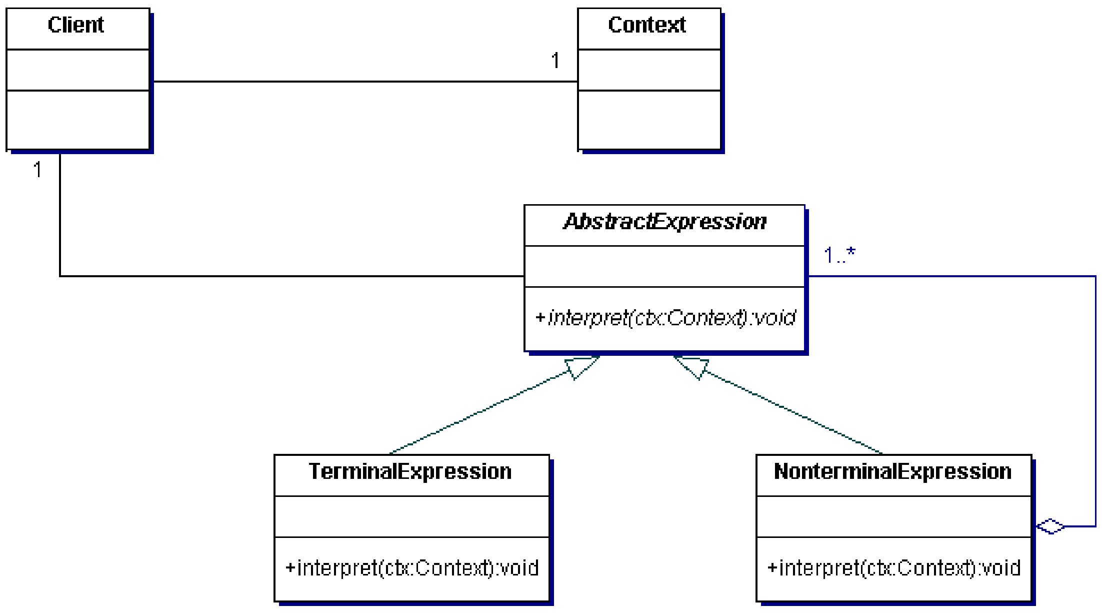
A class of problems occurs repeatedly in a well-defined and well-understood domain. If the domain were characterized with a "language", then problems could be easily solved with an interpretation "engine".

Discussion

The Interpreter pattern discusses: defining a domain language (i.e. problem characterization) as a simple language grammar, representing domain rules as language sentences, and interpreting these sentences to solve the problem. The pattern uses a class to represent each grammar rule. And since grammars are usually hierarchical in structure, an inheritance hierarchy of rule classes maps nicely. An abstract base class specifies the method interpret(). Each concrete subclass implements interpret() by accepting (as an argument) the current state of the language stream, and adding its contribution to the problem solving process.

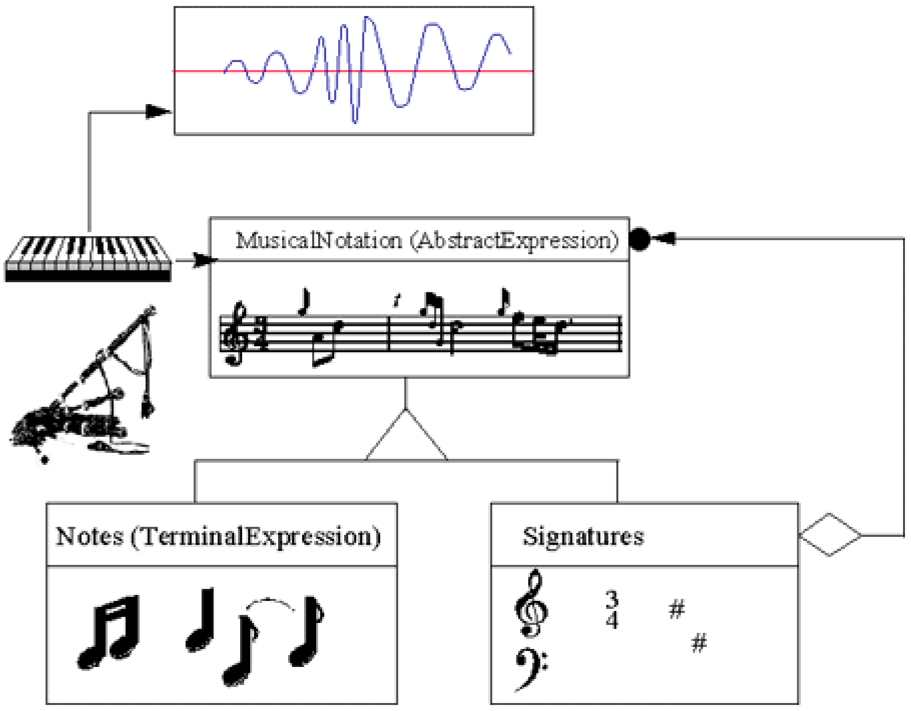
Structure




Client
Context
1
1
AbstractExpression
+interpret(ctx:Context):void
1..*
TerminalExpression
+interpret(ctx:Context):void
NonterminalExpression
+interpret(ctx:Context):void

Example

The Interpreter pattern defines a grammatical representation for a language and an interpreter to interpret the grammar. Musicians are examples of Interpreters. The pitch of a sound and its duration can be represented in musical notation on a staff. This notation provides the language of music. Musicians playing the music from the score are able to reproduce the original pitch and duration of each sound represented. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]


MusicalNotation (AbstractExpression)
Notes (TerminalExpression)
Signatures

Rules of thumb

Interpreter can use State to define parsing contexts. [GOF, p349]

The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable). [GOF, p255]

Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight. [GOF, p255]

Program Examples

Example 1:

```
// Purpose.  Interpreter
//
// Define a grammar for a language, and map each rule to a class.

#include <iostream.h>
#include <string.h>

int sub(char* i, char* o, char* c)
{
strcat(o, c);  strcpy(i, &(i[1]));
return 1;
 }

class A
{
public:
```

```
int eval( char* i, char* o )
          {
                    if (i[0] == 'a')
                    return sub(i,o,"1");
                    return 0;
          }
};

class B
{
 public:
int eval( char* i, char* o )
          {
          if (i[0] == 'b')
                    return sub(i,o,"2");
          return 0;
           }
};

class C
{
public:
int eval( char* i, char* o )
          {
                    if (i[0] == 'c')
                              return sub(i,o,"3");
                    return 0;
          }
};

class D
{
public:
int eval( char* i, char* o )
          {
                    if (i[0] == 'd')
                              return sub(i,o,"4");
                    return 0;
          }
};

class E
{
public:
int eval( char* i, char* o )
          {
                    if (i[0] == 'e')
                              return sub(i,o,"5");
                    return 0;
          }
};
```

```
class F
{
public:
int eval( char* i, char* o )
        {
                if (i[0] == 'f')
                        return sub(i,o,"6");
                return 0;
        }
};

class G
{
public:
int eval( char* i, char* o )
        {
                if (i[0] == 'g')
                        return sub(i,o,"7");
                return 0;
        }
};

class H
{
public:
int eval( char* i, char* o )
        {
                if (i[0] == 'h')
                        return sub(i,o,"8");
                return 0;
        }
};

class Arti
{
public:
int eval( char* i, char* o )
        {
                if (a.eval(i,o)) return 1;
                if (b.eval(i,o)) return 1;
                return 0;
         }
private:
        A a;  B b;
};

class Noun
{
public:
int eval( char* i, char* o )
        {
                if (c.eval(i,o)) return 1;
                if (d.eval(i,o)) return 1;
                return 0;
        }
private:
C c;  D d;
};

class Acti
{
public:
int eval( char* i, char* o )
        {
                if (e.eval(i,o)) return 1;
                if (f.eval(i,o)) return 1;
                return 0;
        }
private:
E e;  F f;
};

class Pass
{
public:
int eval( char* i, char* o )
        {
                if (g.eval(i,o)) return 1;
                if (h.eval(i,o)) return 1;
                return 0;
        }
private:
G g;  H h;
};

class NP
{
public:
int eval( char* i, char* o )
{
        if (arti.eval(i,o))
        if (noun.eval(i,o)) return 1;
        return 0;
}
private:
        Arti arti;  Noun noun;
};

class Verb
{
public:
int eval( char* i, char* o )
        {
                if (acti.eval(i,o)) return 1;
                if (pass.eval(i,o)) return 1;
                return 0;
        }
private:
        Acti acti;  Pass pass;
};
```

```
class Sent
{
public:
int eval( char* i, char* o )
        {
                if (np.eval(i,o))
                if (verb.eval(i,o)) return 1;
                return 0;
        }
private:
        NP np;  Verb verb;
};

void main( void )
{
Sent S;  char* t[] = {"ace","bdh","abc","ceg","bcfgh"};
char  i[10], o[10];
for (int j=0; j < 5; j++)
         {
        strcpy(i,t[j]); strcpy(o,"");
        cout << i << " is ";
        if ( ! S.eval(i,o) || i[0])
                cout << "bad" << endl;
        else
        cout << o << endl;
        }
}

// ace is 135
// bdh is 248
// abc is bad
// ceg is bad
// bcfgh is bad
```

Example 2:

```
// Purpose.  Interpreter design pattern demo (with Template Method)
//
// Discussion.  Uses a class hierarchy to represent the grammar given
// below.  When a roman numeral is provided, the class hierarchy validates
// and interprets the string.  RNInterpreter "has" 4 sub-interpreters.
// Each sub-interpreter receives the "context" (remaining unparsed string
// and cumulative parsed value) and contributes its share to the processing.
// Sub-interpreters simply define the Template Methods declared in the base
// class RNInterpreter.
//   romanNumeral ::= {thousands} {hundreds} {tens} {ones}
//   thousands, hundreds, tens, ones ::= nine | four | {five} {one} {one} {one}
//   nine ::= "CM" | "XC" | "IX"
//   four ::= "CD" | "XL" | "IV"
//   five ::= 'D' | 'L' | 'V'
//   one  ::= 'M' | 'C' | 'X' | 'I'

#include <iostream.h>
#include <string.h>
```

```
class Thousand;  class Hundred;  class Ten;  class One;

class RNInterpreter
{
public:
        RNInterpreter();                    // ctor for client
        RNInterpreter( int ) { }            // ctor for subclasses, avoids infinite loop
        int interpret( char* );     // interpret() for client
        virtual void interpret( char* input, int& total ) // for internal use
        {
                int  index;  index = 0;
                if ( ! strncmp(input, nine(), 2))
                {
                        total += 9 * multiplier();
                        index += 2;
                 }
                else if ( ! strncmp(input, four(), 2))
                 {
                        total += 4 * multiplier();
                        index += 2;
                }
                else
                {
                        if (input[0] == five())
                        {
                                total += 5 * multiplier();
                                index = 1;
                        }
                        else index = 0;
                        for (int end = index + 3 ; index < end; index++)
                                if (input[index] == one())
                                        total += 1 * multiplier();
                                else break;
                }
                strcpy( input, &(input[index]));
         }      // remove leading chars processed

protected:
// cannot be pure virtual because client asks for instance
        virtual char  one()  { }
        virtual char* four() { }
        virtual char  five()  { }
        virtual char* nine() { }
        virtual int   multiplier() { }
private:
        RNInterpreter*  thousands;
        RNInterpreter*  hundreds;
        RNInterpreter*  tens;
        RNInterpreter*  ones;
};

class Thousand : public RNInterpreter
{
public:
```

```
// provide 1-arg ctor to avoid infinite loop in base class ctor
        Thousand( int ) : RNInterpreter(1) { }
protected:
        char  one()      { return 'M'; }
        char* four()     { return ""; }
        char  five()     { return '\0'; }
        char* nine()     { return ""; }
        int   multiplier() { return 1000; }
};

class Hundred : public RNInterpreter
{
public:
        Hundred( int ) : RNInterpreter(1) { }
protected:
        char  one()       { return 'C'; }
        char* four() { return "CD"; }
        char  five()      { return 'D'; }
        char* nine() { return "CM"; }
        int   multiplier() { return 100; }
};

class Ten : public RNInterpreter
{
public:
        Ten( int ) : RNInterpreter(1) { }
protected:
        char  one()      { return 'X'; }
        char* four() { return "XL"; }
        char  five()      { return 'L'; }
        char* nine() { return "XC"; }
        int   multiplier() { return 10; }
};

class One : public RNInterpreter
{
public:
        One( int ) : RNInterpreter(1) { }
protected:
        char  one()      { return 'I'; }
        char* four()     { return "IV"; }
        char  five()     { return 'V'; }
        char* nine()     { return "IX"; }
        int   multiplier() { return 1; }
};

RNInterpreter::RNInterpreter()
{
// use 1-arg ctor to avoid infinite loop
        thousands = new Thousand(1);
        hundreds = new Hundred(1);
        tens      = new Ten(1);
        ones      = new One(1);
}
```

```
int RNInterpreter::interpret( char* input )
{
        int  total;  total = 0;
        thousands->interpret( input, total );
        hundreds->interpret( input, total );
        tens->interpret( input, total );
        ones->interpret( input, total );
        if (strcmp(input, ""))                  // if input was invalid, return 0
                return 0;
        return total;
}


void main()
{
        RNInterpreter interpreter;
        char        input[20];
        cout << "Enter Roman Numeral: ";
        while (cin >> input)
        {
                cout << "   interpretation is " << interpreter.interpret(input) << endl;
                cout << "Enter Roman Numeral: ";
        }
}

// Enter Roman Numeral: MCMXCVI
//    interpretation is 1996
// Enter Roman Numeral: MMMCMXCIX
//    interpretation is 3999
// Enter Roman Numeral: MMMM
//    interpretation is 0
// Enter Roman Numeral: MDCLXVIIII
//    interpretation is 0
// Enter Roman Numeral: CXCX
//    interpretation is 0
// Enter Roman Numeral: MDCLXVI
//    interpretation is 1666
// Enter Roman Numeral: DCCCLXXXVIII
//    interpretation is 888
```

Iterator

Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Problem

Need to "abstract" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently.

Discussion

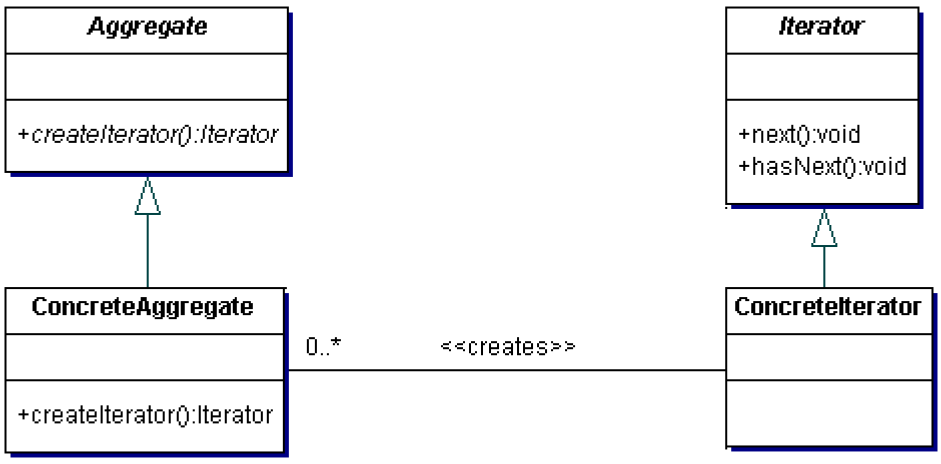
"An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you need to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you'll require. You might also need to have more than one traversal pending on the same list." [GOF, p257] And, providing a uniform interface for traversing many types of aggregate objects (i.e. polymorphic iteration) might be valuable.

The Iterator pattern lets you do all this. The key idea is to take the responsibility for access and traversal out of the aggregate object and put it into an Iterator object that defines a standard traversal protocol.

The Iterator abstraction is fundamental to an emerging technology called "generic programming". This strategy seeks to explicitly separate the notion of "algorithm" from that of "data structure". The motivation is to: promote component-based development, boost productivity, and reduce configuration management.

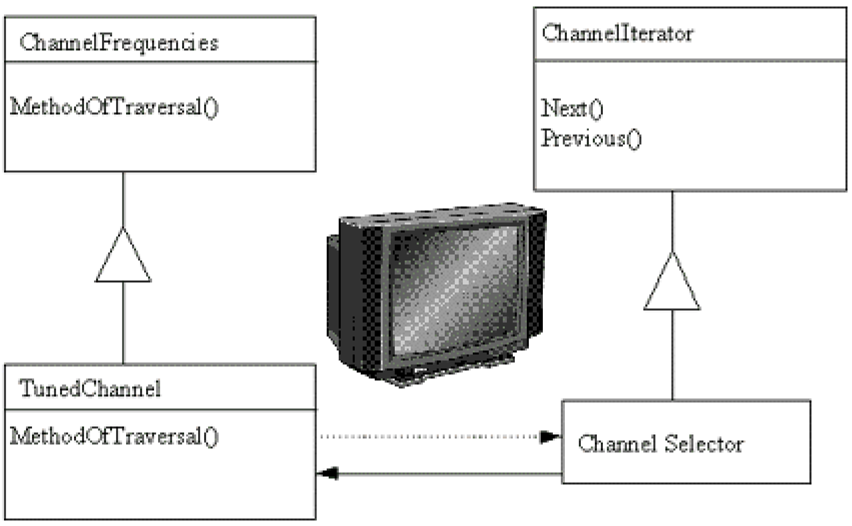
As an example, if you wanted to support four data structures (array, binary tree, linked list, and hash table) and three algorithms (sort, find, and merge), a traditional approach would require four times three permutations to develop and maintain. Whereas, a generic programming approach would only require four plus three configuration items.

Structure



Example

The Iterator provides ways to access elements of an aggregate object sequentially without exposing the underlying structure of the object. Files are aggregate objects. In office settings where access to files is made through administrative or secretarial staff, the Iterator pattern is demonstrated with the secretary acting as the Iterator. Several television comedy skits have been developed around the premise of an executive trying to understand the secretary's filing system. To the executive, the filing system is confusing and illogical, but the secretary is able to access files quickly and efficiently. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable). [GOF, p255]

Iterator can traverse a Composite. Visitor can apply an operation over a Composite. [GOF, p173]

Polymorphic Iterators rely on Factory Methods to instantiate the appropriate Iterator subclass. [GOF, p271]

Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally. [GOF, p271]

Program Examples

Example 1:

```
// Purpose.  Iterator design pattern

// Take traversal-of-a-collection functionality out of the collection and
// promote it to "full object status".  This simplifies the collection, allows
// many traversals to be active simultaneously, and decouples collection algo-
// rithms from collection data structures.

// "Every 'container-like' class must have an iterator." [Lakos]  It may seem
// like a violation of encapsulation for a Stack class to allow its users to
// access its contents directly, but John Lakos' argument is that the designer
// of a class inevitably leaves something out.  Later, when users need addi-
// tional functionality, if an iterator was originally provided, they can add
// functionality with "open for extension, closed for modification".  Without
// an iterator, their only recourse is to invasively customize production
// code.  Below, the orginal Stack class did not include an equality operator,
// but it did include an iterator.  As a result, the equality operator could
// be readily retrofitted.

// 1. Design an "iterator" class for the "container" class
// 2. Add a createIterator() member to the container class
// 3. Clients ask the container object to create an iterator object
// 4. Clients use the first(), isDone(), next(), and currentItem() protocol

#include <iostream>
using namespace std;

class Stack
{
   int items[10];
   int sp;
public:
   friend class StackIter;
   Stack()            { sp = -1; }
   void push( int in ) { items[++sp] = in; }
   int  pop()          { return items[sp--]; }
   bool isEmpty()      { return (sp == -1); }
   StackIter* createIterator() const;      // 2. Add a createIterator() member
};

class StackIter                        // 1. Design an "iterator" class
 {
   const Stack* stk;
   int index;
public:
   StackIter( const Stack* s ) { stk = s; }
   void first()               { index = 0; }
   void next()                  { index++; }
   bool isDone()               { return index == stk->sp+1; }
   int  currentItem()          { return stk->items[index]; }
};

StackIter* Stack::createIterator() const { return new StackIter( this ); }

bool operator==( const Stack& l, const Stack& r )
 {
   // 3. Clients ask the container object to create an iterator object
   StackIter* itl = l.createIterator();
   StackIter* itr = r.createIterator();
   // 4. Clients use the first(), isDone(), next(), and currentItem() protocol
   for (itl->first(), itr->first(); ! itl->isDone(); itl->next(), itr->next())
            if (itl->currentItem() != itr->currentItem())
                              break;
   bool ans = itl->isDone() && itr->isDone();
   delete itl;
   delete itr;
   return ans;
}

void main( void )
{
   Stack  s1;
   for (int i=1; i < 5; i++) s1.push(i);
   Stack  s2( s1 ), s3( s1 ), s4( s1 ), s5( s1 );
   s3.pop();
   s5.pop();
   s4.push( 2 );
   s5.push( 9 );
   cout << "1 == 2 is "<< (s1 == s2) << endl;
   cout << "1 == 3 is "<< (s1 == s3) << endl;
   cout << "1 == 4 is "<< (s1 == s4) << endl;
   cout << "1 == 5 is "<< (s1 == s5) << endl;
}

// 1 == 2 is 1
// 1 == 3 is 0
// 1 == 4 is 0
// 1 == 5 is 0
```

Example 2:

```
// Purpose.  Iterator using operators instead of methods
//
// Discussion.  John Lakos suggests the GOF and STL iterator interfaces are:
// error-prone (possibility of misspelling method names), clumsy, and require
// too much typing.  This design uses nothing but "intuitive" operators.
// Notice also that no createIterator() was specified.  The user creates these
// iterators as local variables, and no clean-up is necessary.

#include <iostream>
using namespace std;
class Stack
{
   int items[10];
   int sp;
public:
   friend class StackIter;
   Stack()            { sp = -1; }
   void push( int in ) { items[++sp] = in; }
   int  pop()         { return items[sp--]; }
   bool isEmpty()      { return (sp == -1); }
};

class StackIter
{
   const Stack& stk;
   int index;
public:
   StackIter( const Stack& s ) : stk( s ) { index = 0; }
   void operator++() { index++; }
   bool operator()() { return index != stk.sp+1; }
   int  operator*()  { return stk.items[index]; }
};

bool operator==( const Stack& l, const Stack& r )
{
   StackIter itl( l ), itr( r );
   for ( ; itl(); ++itl, ++itr)
      if (*itl != *itr) break;
   return ! itl() && ! itr();
}

void main( void )
{
   Stack  s1;   int  i;
   for (i=1; i < 5; i++) s1.push(i);
   Stack  s2( s1 ), s3( s1 ), s4( s1 ), s5( s1 );
   s3.pop();       s5.pop();
   s4.push( 2 );   s5.push( 9 );
   cout << "1 == 2 is "<< (s1 == s2) << endl;
   cout << "1 == 3 is "<< (s1 == s3) << endl;
   cout << "1 == 4 is "<< (s1 == s4) << endl;
   cout << "1 == 5 is "<< (s1 == s5) << endl;
}
```

```
// 1 == 2 is 1
// 1 == 3 is 0
// 1 == 4 is 0
// 1 == 5 is 0
```

Example 3:

```
// Purpose.  Iterator using the Java interface
//
// Discussion.  Java's standard collection classes have a much leaner inter-
// face.  Their next() methods combine the next() and currentItem() methods.

#include <iostream>
using namespace std;

class Stack
{
   int items[10];
   int sp;
public:
   friend class StackIter;
   Stack()            { sp = -1; }
   void push( int in ) { items[++sp] = in; }
   int  pop()          { return items[sp--]; }
   bool isEmpty()      { return (sp == -1); }
   StackIter* iterator() const;
};

class StackIter
{
   const Stack* stk;
   int index;
public:
   StackIter( const Stack* s ) { stk = s;  index = 0; }
   int  next()                { return stk->items[index++]; }
   bool hasNext()             { return index != stk->sp+1; }
};

StackIter* Stack::iterator() const
{
return new StackIter( this );
}

bool operator==( const Stack& l, const Stack& r )
        {
                 StackIter* itl = l.iterator();
                StackIter* itr = r.iterator();
                while (itl->hasNext())
                        if (itl->next() != itr->next())
                        {
                              delete itl;
                              delete itr;
                              return false;
```

```
                                }
                        bool ans = (! itl->hasNext()) && ( ! itr->hasNext());
                        delete itl;
                        delete itr;
                        return ans;
            }

void main( void )
{
   Stack  s1;   int  i;
   for (i=1; i < 5; i++) s1.push(i);
   Stack  s2( s1 ), s3( s1 ), s4( s1 ), s5( s1 );
   s3.pop();
   s5.pop();
   s4.push( 2 );
   s5.push( 9 );
   cout << "1 == 2 is "<< (s1 == s2) << endl;
   cout << "1 == 3 is "<< (s1 == s3) << endl;
   cout << "1 == 4 is "<< (s1 == s4) << endl;
   cout << "1 == 5 is "<< (s1 == s5) << endl;
}

// 1 == 2 is 1
// 1 == 3 is 0
// 1 == 4 is 0
// 1 == 5 is 0
```

Example 4:

```
// Purpose.  Simple implementation of an Iterator (uses parallel dynamic array)
//
// Discussion.  Instead of having to write an involved stack-like solution to
// handle step-wise recursive descent, use a little extra memory to maintain a
// sequential representation of the original hierarchical data.  The replicated
// data are not Node objects, they are lightweight pointers.  The array is
// initialized using a recursive method similar to traverse(Node*).

#include <iostream>
#include <ctime>
using namespace std;

class BST
{
   friend class Iterator;
   struct Node
            {
            Node( int );
              int value;
            Node* left;
             Node* rite;
            };
   Node* root;
   int   size;
   void add( Node**, int );
   void traverse( Node* );
public:
   BST() { root = 0;  size = 0; }
   void add( int );
   void traverse();
   Iterator* createIterator() const;
};

class Iterator
{
   const BST* tree;
   BST::Node** array;
   int index;
   void populateArray( BST::Node* current )
            {
              if (current->left)
                        populateArray( current->left );
            array[index++] = current;
            if (current->rite)
                        populateArray( current->rite );
            }
public:
   Iterator( const BST* s )
            {
                  tree = s;
                  array = 0;  index = 0;
               }
   ~Iterator() { delete [] array; }
   void first()
             {
                  delete [] array;
                  array = new BST::Node*[tree->size];
                  index = 0;
                  populateArray( tree->root );
                  index = 0;
             }
   void next()              { index++; }
   bool isDone()         { return index == tree->size; }
   BST::Node* currentItem() { return array[index]; }
};

void main( void )
{
   srand( time( 0 ) );
   BST  tree;
   for (int i=0; i < 20; i++)
                        tree.add( rand() % 20 + 1 );
   cout << "traverse: ";
   tree.traverse();
   cout << "\niterator: ";
   Iterator* it = tree.createIterator();
   for (it->first(); ! it->isDone(); it->next())
                  cout << it->currentItem()->value << ' ';
   cout << "\niterator: ";
   for (it->first(); ! it->isDone(); it->next())
```

```
            cout << it->currentItem()->value << ' ';
  cout << '\n';
}

// traverse: 1 2 3 7 8 9 9 10 11 11 13 14 14 14 15 17 18 19 19 20
// iterator: 1 2 3 7 8 9 9 10 11 11 13 14 14 14 15 17 18 19 19 20
// iterator: 1 2 3 7 8 9 9 10 11 11 13 14 14 14 15 17 18 19 19 20


BST::Node::Node( int v ) { value = v;  left = rite = 0; }

void BST::add( Node** n, int v )
{
  if ( ! *n)
          {
          *n = new Node( v );
              size++;
          }
  else if (v <= (*n)->value)
        add( &((*n)->left), v );
  else
           add( &((*n)->rite), v );
}

void BST::add( int v ) { add( &root, v ); }
void BST::traverse() { traverse( root ); }

void BST::traverse( Node* n )
{
  if (n->left)
        traverse( n->left );
  cout << n->value << ' ';
  if (n->rite)
        traverse( n->rite );
}

Iterator* BST::createIterator() const
{
      return new Iterator( this );
}
```

Example 5:

```
// Purpose.  Fairly reusable iterator for step-wise recursive descent
//
// Discussion.  The Composite/Component/Leaf code is one of the previous
// Composite demos.  Methods added were: Component::outputValue() and
// Composite::createIterator().

#include <iostream>
#include <vector>
using namespace std;
```

```
class Component
{
public:
   virtual void traverse() = 0;
   virtual void outputValue() { }
};

class Leaf : public Component
{
   int value;
public:
   Leaf( int val ) { value = val; }
   /*virtual*/ void traverse() { cout << value << ' '; }
   /*virtual*/ void outputValue() { traverse(); }
};

class Composite : public Component
{
   vector<Component*> children;
public:
   friend class Iterator;
   void add( Component* ele ) { children.push_back( ele ); }
   /*virtual*/ void traverse()
         {
                  for (int i=0; i < children.size(); i++)
                           children[i]->traverse();
         }
   Iterator* createIterator();
};

class Memento
{
public:
   struct MgrIdx
         {
                  MgrIdx( Composite* m=0, int i=0 )
                           {
                                    mgr = m;
                                    idx = i;
                           }
                  Composite* mgr;
                   int        idx;
          };

   void initialize( Composite* root )
         {
                  vec.resize( 0 );
                  vec.push_back( MgrIdx( root, 0 ) );
         }
   bool isEmpty()        { return vec.size() == 0; }
   void push( MgrIdx ds ) { vec.push_back( ds ); }
   MgrIdx top()          { return vec.back(); }
MgrIdx pop()
            {
               MgrIdx ds = vec.back();
               vec.pop_back();
              return ds;
               }
private:
   vector<MgrIdx> vec;
};

// Dependencies on actual class playing the role of "Composite":
//    Composite class name, children attribute name
class Iterator
{
   Composite* root;
   Memento    memento;
   bool       done;
public:
   Iterator( Composite* rooot ) { root = rooot; }
   void first()
   {
      memento.initialize( root );
      done = false;
   }

   void next()
    {
      Memento::MgrIdx ds = memento.pop();
      ds.idx++;
      // if (and while) you are at end-of-composite, go up
      while (ds.idx == ds.mgr->children.size())
            {
              if (memento.isEmpty()) { done = true;  return; }
             ds = memento.pop();
            ds.idx++;
            }
      memento.push( ds );
      Composite* compo;
      if (compo = dynamic_cast<Composite*>(ds.mgr->children[ds.idx]))
             memento.push( Memento::MgrIdx( compo, 0 ) );
   }

   Component* currentItem()
            {
             Memento::MgrIdx ds = memento.top();
            return ds.mgr->children[ds.idx];
            }

   bool isDone() { return done; }
};

Iterator* Composite::createIterator() { return new Iterator( this ); }
```

```
void main( void )
{
   Composite containers[4];
   for (int i=0; i < 4; i++)
            for (int j=0; j < 3; j++)
                     containers[i].add( new Leaf( i * 3 + j ) );

   for (i=1; i < 4; i++)
                    containers[0].add( &(containers[i]) );

   cout << "traverse: ";
   containers[0].traverse();

   Iterator* it = containers[0].createIterator();
   cout << "\niterator: ";

   for (it->first(); ! it->isDone(); it->next())
              it->currentItem()->outputValue();

   cout << '\n';
   cout << "iterator: ";

   for (it->first(); ! it->isDone(); it->next())
              it->currentItem()->outputValue();

   cout << '\n';
   delete it;
}

// traverse: 0 1 2 3 4 5 6 7 8 9 10 11
// iterator: 0 1 2 3 4 5 6 7 8 9 10 11
// iterator: 0 1 2 3 4 5 6 7 8 9 10 11
```

Example 6:

```
// Purpose.  Iterator design pattern lab
//
// Problem.  The BST class can traverse itself, but, the client of the BST
// cannot stop the traversal at will and interject application-specific
// functionality.  An Iterator class is a "partner" to a "container-like"
// class that provides the level of access and control that a client might
// need, while still maintaining the encapsulation of the original class.
// John Lakos has asserted that the interface of a "container-like" class
// is not "necessary and sufficient" until an Iterator capability is provided.
//
// Assignment.
//  Declare a class Iterator
//  The public interface of class Iterator should consist of -
//      Iterator( BST* );
//      void first();
//      void next();
//      int isDone();
//      Node& currentItem();
//  Make class Iterator a friend of class BST
//  Add the following method to class BST
//     Iterator* createIterator();
//  Add the following to the end of main() -
//      create an Iterator object
//      in a for loop, traverse the BST by using Iterator's first(), next(),
//         isDone(), and currentItem() methods
//      in a second for loop, traverse the BST again
//      deallocate the Iterator object
//  The BST::createIterator() implementation needs to create and return an
//   "appropriately initialized" instance of class Iterator
//  The school solution private data members of class Iterator are:
//      BST*    bst;
//      Node**  list;
//      int     nextNode;
//  A strategy for the Iterator::first() implementation is -
//      create a dynamic array of type Node* whose size is equal to the number
//         of nodes in the BST
//      traverse the BST, copying each Node* into the dynamic array (the array
//         will now "mirror" the BST)
//      initialize a "next array element" data member to 0
//  Traversing the BST to initialize the dynamic array could be implemented
//   by defining a recursive function that looks a lot like the current
//   BST::traverse(Node*) method.  The school solution implementation defines
//   the following private utility method -
//      void buildList( Node* current ) {
//         if (current->left != 0) buildList( current->left );
//         list[nextNode++] = current;
//         if (current->right != 0) buildList( current->right );
//      }
//  Iterator::next() simply increments the "next array element" data member
//  Iterator::currentItem() uses the "next array element" data member to
//   index into the dynamic array and then returns the dereferenced pointer
//  Iterator::isDone() needs an implementation
//  The dynamic array created in Iterator::first() needs to be deallocated if
//   a second call to Iterator::first() is made

#include <iostream.h>
#include <stdlib.h>
#include <time.h>

struct Node
{
   int value;  Node* left;  Node* right;
   Node()
         {
              left = right = 0;
          }
   friend ostream& operator<< ( ostream& os, Node& n ) { return os << n.value; }
};

class BST
{
private:
   Node*  root;
   int    size;
public:
   BST()
   {
     root = 0;
   }
   void add( int in )
         {
             if (root == 0)
                 {
                       root = new Node;
                       root->value = in;
                       size = 1;
                       return;
                  }
             add( in, root );
           }

void traverse() { traverse( root ); }

private:
   void add( int in, Node* current )
         {
              if (in < current->value)
                       if (current->left == 0)
                         {
                                 current->left = new Node();
                                 current->left->value = in;
                                 size++;
                         }
                       else
                                 add( in, current->left );
         else
               if (current->right == 0)
                 {
                           current->right = new Node();
                           current->right->value = in;
                           size++;
                 }
              else
                        add( in, current->right );
         }

  void traverse( Node* current )
         {
              if (current->left != 0)
                  traverse( current->left );
              cout << current->value << "  ";
              if (current->right != 0)
                  traverse( current->right );
         }
};
```

```
void main( void )
{
   BST     bst;
   time_t  t;
   srand((unsigned) time(&t));

   cout << "original:  ";
   for (int i=0, val; i < 15; i++)
           {
                 val = rand() % 49 + 1;
                 cout << val << "  ";
                bst.add( val );
           }
   cout << "\ntraverse:  ";
   bst.traverse();
   cout << endl;
}

// original:  11  43  7  2  22  3  25  40  41  36  32  11  24  11  37
// traverse:  2  3  7  11  11  11  22  24  25  32  36  37  40  41  43
// Iterator:  2  3  7  11  11  11  22  24  25  32  36  37  40  41  43
// Iterator:  2  3  7  11  11  11  22  24  25  32  36  37  40  41  43
```

Mediator

Intent

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Problem

We want to design reusable components, but dependencies between the potentially reusable pieces demonstrate the "spaghetti code" phenomenon (trying to scoop a single serving results in an "all or nothing clump").

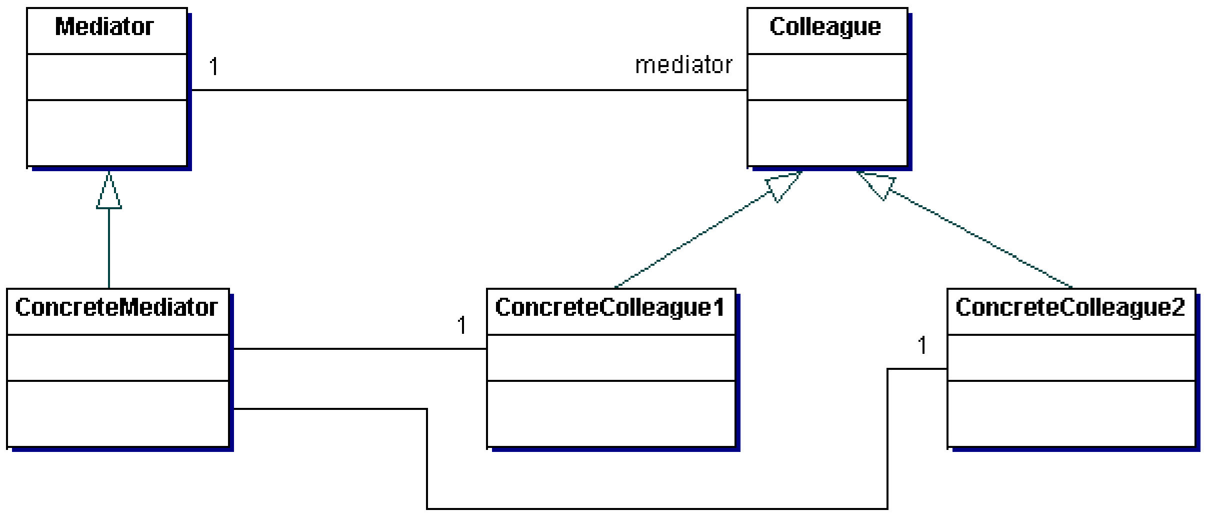
Discussion

Partitioning a system into many objects generally enhances reusability, but proliferating interconnections between those objects tend to reduce it again. The mediator object: encapsulates all interconnections, acts as the hub of communication, is responsible for controlling and coordinating the interactions of its clients, and promotes loose coupling by keeping objects from referring to each other explicitly.

The Mediator pattern promotes a "many-to-many relationship network" to "full object status". Modelling the inter-relationships with an object enhances encapsulation, and allows the behavior of those inter-relationships to be modified or extended through subclassing.

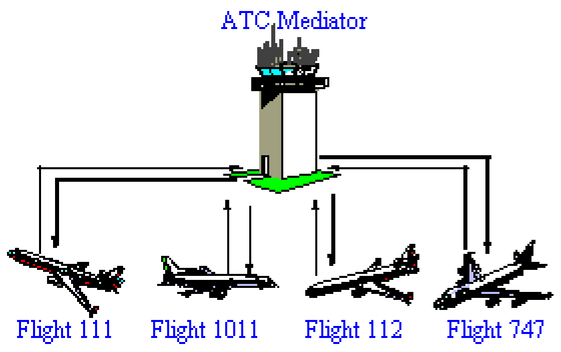
An example where Mediator is useful is the design of a user and group capability in an operating system. A group can have zero or more users, and, a user can be a member of zero or more groups. The Mediator pattern provides a flexible and non-invasive way to associate and manage users and groups.

Structure



Example

The Mediator defines an object that controls how a set of objects interact. Loose coupling between colleague objects is achieved by having colleagues communicate with the Mediator, rather than with each other. The control tower at a controlled airport demonstrates this pattern very well. The pilots of the planes approaching or departing the terminal area communicate with the tower rather than explicitly communicating with one another. The constraints on who can take off or land are enforced by the tower. It is important to note that the tower does not control the whole flight. It exists only to enforce constraints in the terminal area. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers. Command normally specifies a sender-receiver connection with a subclass. Mediator has senders and receivers reference each other indirectly. Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time. [GOF, p347]

Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators. [GOF, p346]

On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them. [GOF, p282]

Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communication between colleague objects, it routinely "adds value", and it is known/referenced by the colleague objects (i.e. it defines a multidirectional protocol). In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes (i.e. it defines a unidirectional protocol where it makes requests of the subsystem classes but not vice versa). [GOF, p193]

PROGRAM EXAMPLES

Example 1:

```
// Purpose.  Mediator
//
// Discussion.  On the left: Node objs interact directly with each other,
// recursion is required, removing a Node is hard, and it is not possi-
// ble to remove the first node.  On the right: a "mediating" List class
// focuses and simplifies all the administrative responsibilities, and
// the recursion (which does not scale up well) has been eliminated.

#include <iostream.h>

class Node
{
public:
		Node( int v, Node* n )
		{
			val_ = v;
			next_ = n;
		}
		void traverse()
		{
			cout << val_ << "  ";
			if (next_)
				next_->traverse();
			else
				cout << endl;
		}
		void removeNode( int v )
		{
			Node*  ptr = (Node*) 1;
			removeNode_( v, &ptr );
		}
	private:
		int    val_;
		Node*  next_;
		void removeNode_(int v, Node** n)
		{
			if (val_ == v)
				*n = next_;
			else
			{
				next_->removeNode_( v, n );
				if (*n != (Node*) 1)
				{
					next_ = *n;
					*n = (Node*) 1;
				}
			}
		}
	};

	void main( void )
	{
	Node  fou( 44, 0 );
	Node  thr( 33, &fou );
	Node  two( 22, &thr );
	Node  one( 11, &two );
	one.traverse();
	one.removeNode( 44 );
	one.traverse();
	one.removeNode( 22 );
	one.traverse();
	}

	// 11  22  33  44
	// 11  22  33
	// 11  33
```

Example 2:

```
#include <iostream.h>

class Node
{
public:
        Node( int v ) { val_ = v; }
        int getVal()  { return val_; }
private:
        int  val_;
};

class List
{
public:
        List()
        {
                for (int i=0; i < 10; i++)
                arr_[i] = 0;
                num_ = 0;
        }
        void addNode( Node* n )
        {
                arr_[num_++] = n;
        }
        void traverse()
         {
                for (int i=0; i < num_; i++)
                        cout << arr_[i]->getVal()<< "  ";
                cout << endl;
        }
        void removeNode( int v )
        {
                int  i, j;
                for (i=0; i < num_; i++)
                if (arr_[i]->getVal() == v)
                {
                for (j=i; j < num_; j++)
                        arr_[j] = arr_[j+1];
                num_--;
                break;
        }
}
private:
        Node*  arr_[10];
        int    num_;
};

void main( void )
{
List  lst;
Node  one( 11 ),  two( 22 );
Node  thr( 33 ),  fou( 44 );
lst.addNode( &one );
lst.addNode( &two );
lst.addNode( &thr );
lst.addNode( &fou );
lst.traverse();
lst.removeNode( 44 );
lst.traverse();
lst.removeNode( 11 );
lst.traverse();
}

// 11  22  33  44
// 11  22  33
// 22  33
```

Example 3:

```
// Purpose.  Mediator design pattern demo.
//
// Discussion.  Though partitioning a system into many objects generally
// enhances reusability, proliferating interconnections tend to reduce it
// again.  You can avoid this problem by capsulating the interconnections
// (i.e. the collective behavior) in a separate "mediator" object.  A
// mediator is responsible for controlling and coordinating the
// interactions of a group of objects.  In this example, the dialog box
// object is functioning as the mediator.  Child widgets of the dialog box
// do not know, or care, who their siblings are.  Whenever a simulated
// user interaction occurs in a child widget [Widget::changed()], the
// widget does nothing except "delegate" that event to its parent dialog
// box [_mediator->widgetChanged( this )].
// FileSelectionDialog::widgetChanged() encapsulates all collective
// behavior for the dialog box (it serves as the hub of communication).
// The user may choose to "interact" with a simulated: filter edit field,
// directories list, files list, or selection edit field.

#include <iostream.h>

class FileSelectionDialog;

class Widget
{
public:
	Widget( FileSelectionDialog* mediator, char* name )
	{
		_mediator = mediator;
		strcpy( _name, name);
	}
	virtual void changed();
	virtual void updateWidget() = 0;
	virtual void queryWidget() = 0;
protected:
	char     _name[20];
private:
	FileSelectionDialog* _mediator;
};
```

```
class List : public Widget
{
public:
        List( FileSelectionDialog* dir, char* name ) : Widget( dir, name ) { }
        void queryWidget()  { cout << "   " << _name << " list queried" << endl; }
        void updateWidget() { cout << "   " << _name << " list updated" << endl; }
};

class Edit : public Widget
{
public:
   Edit( FileSelectionDialog* dir, char* name ): Widget( dir, name ) { }
   void queryWidget()  { cout << "   " << _name << " edit queried" << endl; }
   void updateWidget() { cout << "   " << _name << " edit updated" << endl; }
};

class FileSelectionDialog
{
public:
      enum Widgets { FilterEdit, DirList, FileList, SelectionEdit };
      FileSelectionDialog()
      {
            _components[FilterEdit]    = new Edit( this, "filter" );
            _components[DirList]       = new List( this, "dir" );
            _components[FileList]      = new List( this, "file" );
            _components[SelectionEdit] = new Edit( this, "selection" );
      }
      virtual ~FileSelectionDialog();
      void handleEvent( int which )
      {
            _components[which]->changed();
      }
      virtual void widgetChanged( Widget* theChangedWidget )
      {
            if (theChangedWidget == _components[FilterEdit] )
            {
                  _components[FilterEdit]->    queryWidget();
                  _components[DirList]->       updateWidget();
                  _components[FileList]->      updateWidget();
                  _components[SelectionEdit]-> updateWidget();
            }
            else if (theChangedWidget == _components[DirList] )
            {
                  _components[DirList]->       queryWidget();
                  _components[FileList]->      updateWidget();
                  _components[FilterEdit]->    updateWidget();
                  _components[SelectionEdit]-> updateWidget();
            }
            else if (theChangedWidget == _components[FileList] )
            {
                  _components[FileList]->      queryWidget();
                  _components[SelectionEdit]-> updateWidget();
            }
            else if (theChangedWidget == _components[SelectionEdit] )
            {
```

```
                        _components[SelectionEdit]-> queryWidget();
                        cout << "   file opened" << endl;
                }
        }
private:
        Widget* _components[4];
};

        FileSelectionDialog::~FileSelectionDialog()
        {
                for (int i=0; i < 3; i++)
                        delete _components[i];
        }

        void Widget::changed()
        {
                _mediator->widgetChanged( this );
        }


void main()
{
        FileSelectionDialog fileDialog;
        int             i;

        cout << "Exit[0], Filter[1], Dir[2], File[3], Selection[4]: ";
        cin >> i;

        while (i)
        {
                fileDialog.handleEvent( i-1 );
                cout << "Exit[0], Filter[1], Dir[2], File[3], Selection[4]: ";
                cin >> i;
        }
}

// Exit[0], Filter[1], Dir[2], File[3], Selection[4]: 1
//    filter edit queried
//    dir list updated
//    file list updated
//    selection edit updated
// Exit[0], Filter[1], Dir[2], File[3], Selection[4]: 2
//    dir list queried
//    file list updated
//    filter edit updated
//    selection edit updated
// Exit[0], Filter[1], Dir[2], File[3], Selection[4]: 3
//    file list queried
//    selection edit updated
// Exit[0], Filter[1], Dir[2], File[3], Selection[4]: 4
//    selection edit queried
//    file opened
// Exit[0], Filter[1], Dir[2], File[3], Selection[4]: 3
//    file list queried
//    selection edit updated
```

Example 4:

```
// Purpose.  Mediator design pattern lab
//
// Discussion.  In this design, each user "knows" its groups, and each group
// "knows" its users.  The many-to-many mapping between the User class and the
// Group class couples them together.  If we "objectified" this mapping rela-
// tionship several benefits would result: the two classes are no longer
// coupled, many mappings could be supported simply by creating many "mediator"
// instances, and the "mediator" abstraction could be refined through
// inheritance.
//
// Assignment.
// o The new main() and the Mediator implementation are provided.
// o Create a Mediator declaration.  It's state will include: array of 10 ptrs
//   to Group, array of 10 ptrs to User, totalGroups int, totalUsers int, 10x10
//   matrix of int for the "mapping".  It's behavior will include:
//   addUser(User*), addGroup(Group*), map(User*,Group*), reportUsers(Group*),
//   reportGroups(User*)
// o User::addGroup() and Group::addUser() have been moved to Mediator
// o User::groups_ and Group::users_ have been moved to Mediator
// o User and Group ctors need to accept a reference to the Mediator instance
// o User::reportGroups() and Group::reportUsers() now delegate to the Mediator

#include <iostream.h>
#include <string.h>

class Group;

class User
 {
public:
   User( char* );
   const char* getName();
   void addGroup( Group* );
   void reportGroups();
private:
   char    name_[15];
   Group*  groups_[10];
   int     total_;
};

class Group
{
public:
   Group( char* );
   const char* getName();
   void addUser( User* );
   void reportUsers();
private:
   char   name_[15];
   User*  users_[10];
   int    total_;
};
```

```
User::User( char* name )
{
  strcpy( name_, name );
  total_ = 0;
}
const char* User::getName()              { return name_; }
void       User::addGroup( Group* group ) { groups_[total_++] = group; }
void User::reportGroups()
 {
  cout << "   " << name_ << ": ";
  for (int i=0; i < total_; i++)
    cout << ((i == 0) ? "" : ", ") << groups_[i]->getName();
  cout << endl;
}

Group::Group( char* name )
{
  strcpy( name_, name );
  total_ = 0;
}
const char* Group::getName()            { return name_; }
void       Group::addUser( User* user ) { users_[total_++] = user; }
void Group::reportUsers()
{
  cout << "   " << name_ << ": ";
  for (int i=0; i < total_; i++)
    cout << ((i == 0) ? "" : ", ") << users_[i]->getName();
  cout << endl;
}

void main( void )
{
  User*    users[4];
  Group*   groups[4];
  users[0]  = new User(  "Kermit"      );
  users[1]  = new User(  "Gonzo"       );
  users[2]  = new User(  "Fozzy"       );
  users[3]  = new User(  "Piggy"       );
  groups[0] = new Group( "Green"       );
  groups[1] = new Group( "ColdBlooded" );
  groups[2] = new Group( "Male"        );
  groups[3] = new Group( "Muppets"     );

  for (int i=0; i < 4; i++)
    for (int j=0; j <= i; j++)
        {
              groups[i]->addUser( users[j] );
              users[j]->addGroup( groups[i] );
        }
  cout << "Groups:" << endl;
  for (i=0; i < 4; i++) groups[i]->reportUsers();

  cout << "Users:" << endl;
  for (i=0; i < 4; i++) users[i]->reportGroups();
  for (i=0; i < 4; i++)
          {
           delete users[i];  delete groups[i];
          }
}

// Groups:
//    Green: Kermit
//    ColdBlooded: Kermit, Gonzo
//    Male: Kermit, Gonzo, Fozzy
//    Muppets: Kermit, Gonzo, Fozzy, Piggy
// Users:
//    Kermit: Green, ColdBlooded, Male, Muppets
//    Gonzo: ColdBlooded, Male, Muppets
//    Fozzy: Male, Muppets
//    Piggy: Muppets

#if 0
Mediator::Mediator()
{
   totalGroups_ = 0;
   totalUsers_  = 0;
   for (int i=0; i < 10; i++)
      for (int j=0; j < 10; j++)
         mappings_[i][j] = 0;
}

void Mediator::addUser( User* user )
{
   users_[totalUsers_++]   = user;
}

void Mediator::addGroup( Group* group )
{
   groups_[totalGroups_++] = group;
}

void Mediator::map( User* user, Group* group )
{
   int  i, j;
   for (i=0; i < totalGroups_; i++)
      if ( ! strcmp( groups_[i]->getName(), group->getName() ) )
         break;
   for (j=0; j < totalUsers_; j++)
      if ( ! strcmp( users_[j]->getName(), user->getName() ) )
         break;
   mappings_[i][j] = 1;
}

void Mediator::reportUsers( Group* g )
{
   int  i, j, first;
   for (i=0; i < totalGroups_; i++)
      if ( ! strcmp( groups_[i]->getName(), g->getName()))
         break;
   for (j=0, first=1; j < totalUsers_; j++)
      if (mappings_[i][j])
         {
            cout << ((first) ? "" : ", ") << users_[j]->getName();
            first = 0;
         }
   cout << endl;
}

void Mediator::reportGroups( User* u )
{
   int  i, j, first;
   for (j=0; j < totalUsers_; j++)
      if ( ! strcmp( users_[j]->getName(), u->getName()))
         break;
   for (i=0, first=1; i < totalGroups_; i++)
      if (mappings_[i][j])
         {
            cout << ((first) ? "" : ", ") << groups_[i]->getName();
            first = 0;
         }
   cout << endl;
}

void main( void )
{
   User*     users[4];
   Group*    groups[4];
   Mediator  mapping;
   users[0]  = new User(  "Kermit",      mapping );
   users[1]  = new User(  "Gonzo",       mapping );
   users[2]  = new User(  "Fozzy",       mapping );
   users[3]  = new User(  "Piggy",       mapping );
   groups[0] = new Group( "Green",       mapping );
   groups[1] = new Group( "ColdBlooded", mapping );
   groups[2] = new Group( "Male",        mapping );
   groups[3] = new Group( "Muppets",     mapping );

   for (int i=0; i < 4; i++)
         {
           mapping.addUser( users[i] );
           mapping.addGroup( groups[i] );
         }

   for (i=0; i < 4; i++)
      for (int j=0; j <= i; j++)
         mapping.map( users[j], groups[i] );

   cout << "Groups:" << endl;
   for (i=0; i < 4; i++)
         groups[i]->reportUsers();

   cout << "Users:" << endl;
   for (i=0; i < 4; i++)
         users[i]->reportGroups();
```

```
for (i=0; i < 4; i++)
        {
        delete users[i];
        delete groups[i];
        }
}
#endif
```

Memento

Intent

Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later.

Problem

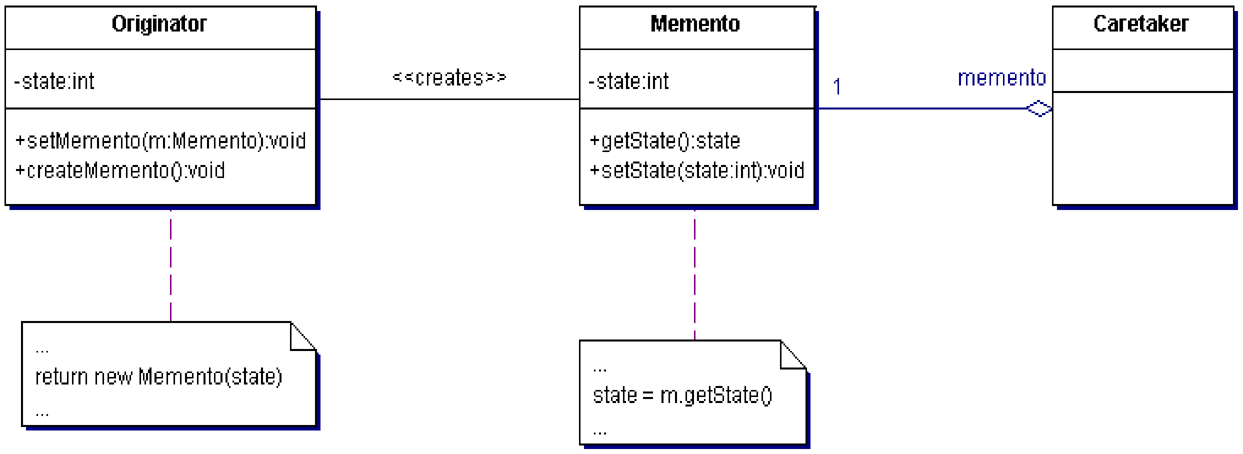
Need to restore an object back to its previous state (e.g. "undo" or "rollback" operations).

Discussion

The client requests a Memento from the source object when it needs to checkpoint the source object's state. The source object initializes the Memento with a characterization of its state. The client is the "care-taker" of the Memento, but only the source object can store and retrieve information from the Memento (the Memento is "opaque" to the client and all other objects). If the client subsequently needs to "rollback" the source object's state, it hands the Memento back to the source object for reinstatement.

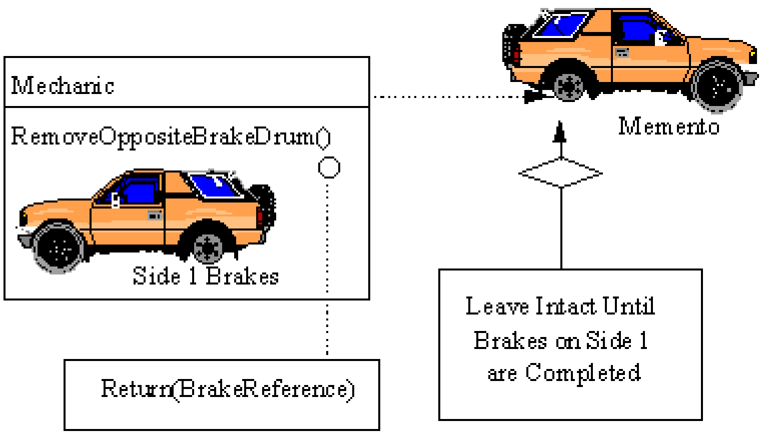
An unlimited "undo" and "redo" capability can be readily implemented with a stack of Command objects and a stack of Memento objects.

Structure



Example

The Memento captures and externalizes an object's internal state so that the object can later be restored to that state. This pattern is common among do-it-yourself mechanics repairing drum brakes on their cars. The drums are removed from both sides, exposing both the right and left brakes. Only one side is disassembled and the other serves as a Memento of how the brake parts fit together. Only after the job has been completed on one side is the other side disassembled. When the second side is disassembled, the first side acts as the Memento. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]


Mechanic
RemoveOppositeBrakeDrum()
Side 1 Brakes
Memento
Leave Intact Until Brakes on Side 1 are Completed
Return(BrakeReference)

Rules of thumb

Command and Memento act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time. Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value. [GOF, p346]

Command can use Memento to maintain the state required for an undo operation. [GOF, 242]

Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally. [GOF, p271]

Program Examples

Example 1:

```
// Purpose.  Memento design pattern

// 1. Assign the roles of "caretaker" and "originator"
// 2. Create a "memento" class and declare the originator a friend
// 3. Caretaker knows when to "check point" the originator
// 4. Originator creates a memento and copies its state to the memento
// 5. Caretaker holds on to (but cannot peek in to) the memento
// 6. Caretaker knows when to "roll back" the originator
// 7. Originator reinstates itself using the saved state in the memento

#include <iostream>
#include <string>
using namespace std;

class Memento                // 2. Create a "memento" class and declare the originator a friend
{
   friend class Stack;
   int *items, num;
   Memento( int* arr, int n )
   {
      items = new int[num = n];
      for (int i=0; i < num; i++) items[i] = arr[i];
   }
public:
   ~Memento() { delete items; }
};

class Stack                  // 1. Stack is the "originator"
{
   int  items[10], sp;
public:
   Stack()              { sp = -1; }
   void     push( int in ) { items[++sp] = in; }
   int      pop()       { return items[sp--]; }
   bool     isEmpty()      { return sp == -1; }
  // 4. Originator creates a memento and copies its state to the memento
   Memento* checkPoint()
   {
      return new Memento( items, sp+1 );
   }
  // 7. Originator reinstates itself using the saved state in the memento
   void rollBack( Memento* m )
   {
      sp = m->num-1;
      for (int i=0; i < m->num; i++) items[i] = m->items[i];
   }
   friend ostream& operator<< ( ostream& os,  const Stack& s )
   {
      string buf( "[ " );
      for (int i=0; i < s.sp+1; i++) { buf += s.items[i]+48;  buf += ' '; }
      buf += ']';
      return os << buf;                          // stack is [ 0 1 2 3 4 ]
   }
};                                               // stack is [ 0 1 2 3 4 5 6 7 8 9 ]
                                                 // popping stack: 9 8 7 6 5 4 3 2 1 0
// 1. main() is the "caretaker"                  // stack is [ ]
void main( void )                                // second is [ 0 1 2 3 4 5 6 7 8 9 ]
{
   Stack s;                                      // first is [ 0 1 2 3 4 ]
   for (int i=0; i < 5; i++)
           s.push( i );                          // popping stack: 4 3 2 1 0
   cout << "stack is " << s << endl;
   Memento* first = s.checkPoint();               // 3. Caretaker knows when to save
   for (i=5; i < 10; i++)
           s.push( i );                           // 5. Caretaker holds on to memento
   cout << "stack is " << s << endl;
   Memento* second = s.checkPoint();              // 3. Caretaker knows when to save
   cout << "popping stack: ";                     // 5. Caretaker holds on to memento
   while ( ! s.isEmpty())
           cout << s.pop() << ' ';
    cout << endl;
   cout << "stack is " << s << endl;
   s.rollBack( second );                          // 6. Caretaker knows when to undo
   cout << "second is " << s << endl;
   s.rollBack( first );                           // 6. Caretaker knows when to undo
   cout << "first is " << s << endl;
   cout << "popping stack: ";
   while ( ! s.isEmpty())
           cout << s.pop() << ' ';
  cout << endl;
 delete first;
 delete second;
}
```

Example 2:

```
// Purpose.  Memento design pattern
//
// Discussion.  A memento is an object that stores a snapshot of the
// internal state of another object.  It can be leveraged to support
// multi-level undo of the Command pattern.  In this example, before a
// command is run against the Number object, Number's current state is
// saved in Command's static memento history list, and the command itself
// is saved in the static command history list.  Undo() simply "pops" the
// memento history list and reinstates Number's state from the memento.
// Redo() "pops" the command history list.  Note that Number's encapsula-
// tion is preserved, and Memento is wide open to Number.

#include <iostream.h>
class Number;

class Memento
{
public:
  Memento( int val ) { _state = val; }
private:
  friend class Number;  // not essential, but p287 suggests this
  int  _state;
};

class Number
{
public:
  Number( int value )                          { _value = value; }
  void dubble()                                { _value = 2 * _value; }
  void half()                                  { _value = _value / 2; }
  int getValue()                               { return _value; }
  Memento* createMemento()                     { return new Memento( _value ); }
  void  reinstateMemento( Memento* mem ) { _value = mem->_state ; }
private:
  int _value;
};

class Command
{
public:
  typedef void (Number::* Action)();
  Command( Number* receiver, Action action )
        {
            _receiver = receiver;
            _action = action;
        }
  virtual void execute()
        {
            _mementoList[_numCommands] = _receiver->createMemento();
            _commandList[_numCommands] = this;
            if (_numCommands > _highWater)
                        _highWater = _numCommands;
            _numCommands++;
            (_receiver->*_action)();
        }
  static void undo()
        {
            if (_numCommands == 0)
                {
                    cout << "*** Attempt to run off the end!! ***" << endl;
                    return;
                }
           _commandList[_numCommands-1]->_receiver->reinstateMemento( mementoList[_numCommands-1] );
            _numCommands--;
        }
  void static redo()
        {
            if (_numCommands > _highWater)
                {
                    cout << "*** Attempt to run off the end!! ***" << endl;
                    return;
                }
            (_commandList[_numCommands]->_receiver->*(_commandList[_numCommands]->_action))();
```

```
                  _numCommands++;
        }
protected:
   Number*       _receiver;
   Action        _action;
  static Command* _commandList[20];
  static Memento* _mementoList[20];
  static int     _numCommands;
  static int     _highWater;
};

Command* Command::_commandList[];
Memento* Command::_mementoList[];
int      Command::_numCommands = 0;
int      Command::_highWater   = 0;

void main()
{
  int i;
  cout << "Integer: ";
  cin >> i;
  Number*   object = new Number(i);

  Command*  commands[3];
  commands[1] = new Command( object, &Number::dubble );
  commands[2] = new Command( object, &Number::half );

  cout << "Exit[0], Double[1], Half[2], Undo[3], Redo[4]: ";
  cin >> i;

  while (i)
  {
    if (i == 3)
      Command::undo();
    else if (i == 4)
      Command::redo();
    else
      commands[i]->execute();
    cout << "   " << object->getValue() << endl;
    cout << "Exit[0], Double[1], Half[2], Undo[3], Redo[4]: ";
    cin >> i;
  }
}

// Integer: 11
// Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 2
//   5
// Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 1
//   10
// Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 2
//   5
// Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 3
//   10
// Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 3
//   5
// Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 3
//    11
// Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 3
// *** Attempt to run off the end!! ***
//    11
// Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 4
//    5
// Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 4
//    10
// Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 4
//    5
// Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 4
// *** Attempt to run off the end!! ***
//    5
```

Example 3:

```
// Purpose.  Memento design pattern lab
//
// Problem.  The GuessGame class expects multiple "clients" to "connect" to it
// with a call to join().  Each client has a different random number chosen
// for it by GuessGame.  In order to uniquely identify itself to the GuessGame
// server, each client passes a string identifier when it calls join() and
// when it calls evaluateGuess().  GuessGame remembers each string identifier
// in an internal table in the join() method, and then searches that table in
// the evaluateGuess() method.  If we put the information that is peculiar to
// each client into a "black box" object (a Memento), and make the client the
// "caretaker" of that object; then the implementation of the server (Guess-
// Game) is greatly simplified.
//
// Assignment.
// o Declare a class Memento
// o Make class GuessGame a "friend" of class Memento
// o Class Memento will have a single private int data member and a private
//   constructor that accepts an int and initializes the data member
// o GuessGame::join() doesn't need to accept a string identifier.  But, it
//   does need to: select a random number, "wrap" that number in a Memento
//   object, and return a pointer to that object.
// o GuessGame::evaluateGuess() will now accept a Memento object instead of
//   a string identifier.  It doesn't need to search its table of string
//   identifiers, and then access its table of random numbers.  Instead, it
//   simply references the Memento's private int data member.
// o All the private data members of class GuessGame can now be retired.
// o Add a Memento* member to struct Game
// o The call to guessServer.join() no longer requires a string argument,
//   and, the Memento* returned from join() needs to be remembered.
// o The call to guessServer.evaluateGuess() now requires a Memento* argument
//   instead of a string argument.
// o Be sure to clean-up the Memento* pointers when you are done
```

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

class GuessGame
{
private:
   int  numbers[10];
   char names[10][20];
   int  total;
public:
   GuessGame()
  {
      time_t  t;
      srand((unsigned) time(&t));
      total = 0;
   }
   void join( char* name )
   {
      strcpy( names[total], name );
      numbers[total++] = rand() % 30 + 1;
   }
   int evaluateGuess( char* name, int guess )
  {
      int i;
      for (i=0; i < total; i++)
         if ( ! strcmp( names[i], name )) break;
      if (guess == numbers[i]) return 0;
      return ((guess > numbers[i]) ? 1 : -1);
   }
};

struct Game
 {
   char name[20];
   int  min, max, done;
   Game() { min = 1;  max = 30;  done = 0;
 }
};

void main( void )
{
   GuessGame  guessServer;
   const int  MAX = 3;
   Game       games[MAX];
   int        gamesComplete = 0;
   int        guess, ret;

   for (int i=0; i < MAX; i++)
   {
      cout << "Enter name: " ;
      cin >> games[i].name;
      guessServer.join( games[i].name );
   }
   while (gamesComplete != MAX)
   {
      for (int j=0; j < MAX; j++)
       {
         if (games[j].done)
                   continue;
         cout << games[j].name <<" ("<<games[j].min <<'-'<<games[j].max <<"): ";
         cin >> guess;
         ret = guessServer.evaluateGuess( games[j].name, guess );
         if (ret == 0)
         {
            cout << "   lights!!  sirens!!  balloons!!" << endl;
            games[j].done = 1;
            gamesComplete++;
         }
         else if (ret < 0)
         {
            cout << "   too low" << endl;
            games[j].min = guess;
         }
         else
         {
            cout << "   too high" << endl;
            games[j].max = guess;
         }
    }
  }
  }
// Enter name: Tom
// Enter name: Dick
// Enter name: Harry
// Tom (1-30): 10
//    too low
// Dick (1-30): 15
//    too high
// Harry (1-30): 20
//    too high
// Tom (10-30): 22
//    too high
// Dick (1-15): 8
//    lights!!  sirens!!  balloons!!
// Harry (1-20): 10
//    too low
// Tom (10-22): 16
//    too high
// Harry (10-20): 15
//    too low
// Tom (10-16): 13
//    too high
// Harry (15-20): 17
//    too low
// Tom (10-13): 11
//    lights!!  sirens!!  balloons!!
// Harry (17-20): 18
//    lights!!  sirens!!  balloons!!
```

Observer

Intent

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

Problem

A large monolithic design does not scale well as new graphing or monitoring requirements are levied.

Discussion

Define an object that is the "keeper" of the data model and/or business logic (the Subject). Delegate all "view" functionality to decoupled and distinct Observer objects. Observers register themselves with the Subject as they are created. Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.

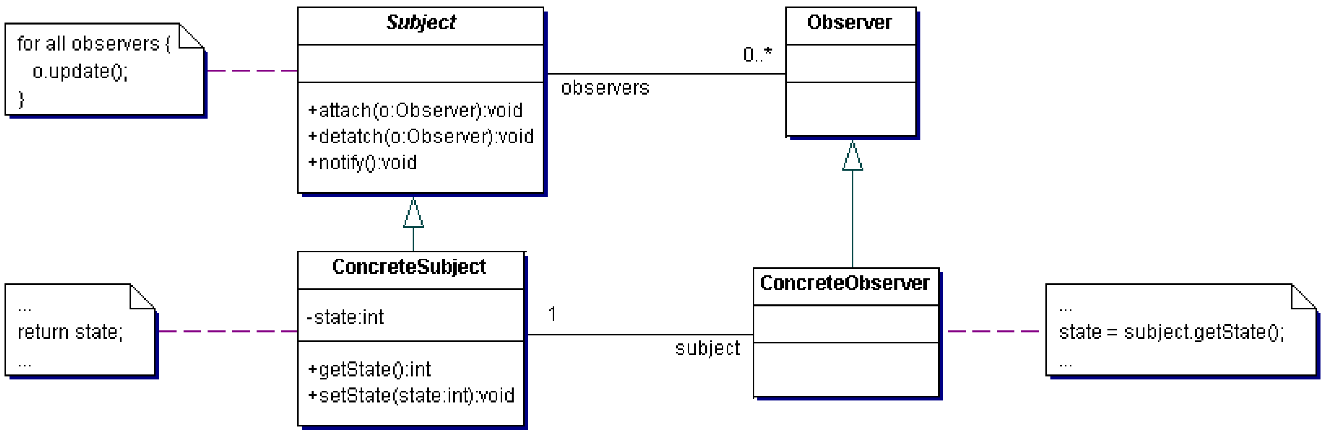
This allows the number and "type" of "view" objects to be configured dynamically, instead of being statically specified at compile-time.

The protocol described above specifies a "pull" interaction model. Instead of the Subject "pushing" what has changed to all Observers, each Observer is responsible for "pulling" its particular "window of interest" from the Subject. The "push" model compromises reuse, while the "pull" model is less efficient.

Issues that are discussed, but left to the discretion of the designer, include: implementing event compression (only sending a single change broadcast after a series of consecutive changes has occurred), having a single Observer monitoring multiple Subjects, and ensuring that a Subject notify its Observers when it is about to go away.

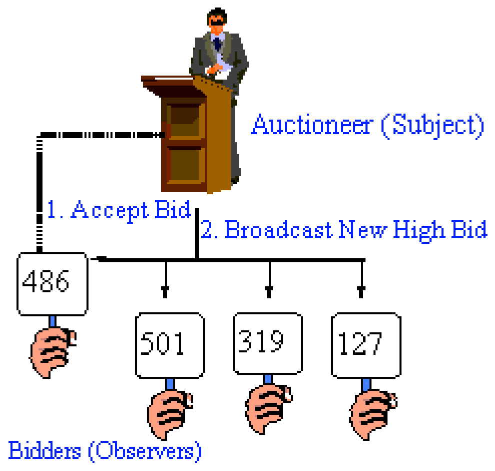
The Observer pattern captures the lion's share of the Model-View-Controller architecture that has been a part of the Smalltalk community for years.

Structure



Example

The Observer defines a one-to-many relationship so that when one object changes state, the others are notified and updated automatically. Some auctions demonstrate this pattern. Each bidder possesses a numbered paddle that is used to indicate a bid. The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid. The acceptance of the bid changes the bid price which is broadcast to all of the bidders in the form of a new bid. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers. Command normally specifies a sender-receiver connection with a subclass. Mediator has senders and receivers reference each other indirectly. Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time. [GOF, p347]

Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators. [GOF, p346]

On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them. [GOF, p282]

Program Examples

```
// Purpose.  Observer design pattern

// 1. Model the "independent" functionality with a "subject" abstraction
// 2. Model the "dependent" functionality with "observer" hierarchy
// 3. The Subject is coupled only to the Observer base class
// 4. Observers register themselves with the Subject
// 5. The Subject broadcasts events to all registered Observers
// 6. Observers "pull" the information they need from the Subject
// 7. Client configures the number and type of Observers

#include <iostream>
#include <vector>
using namespace std;

class Subject                                // 1. "independent" functionality
{
   vector<class Observer*> views;    // 3. Coupled only to "interface"
   int value;
public:
   void attach( Observer* obs ) { views.push_back( obs ); }
   void setVal( int val )       { value = val;  notify(); }
   int  getVal()                { return value; }
   void notify();
};
class Observer                               // 2. "dependent" functionality
{
   Subject* model;
   int      denom;
public:
   Observer( Subject* mod, int div )
   {
      model = mod;  denom = div;
      // 4. Observers register themselves with the Subject
      model->attach( this );
   }
   virtual void update() = 0;
protected:
   Subject* getSubject() { return model; }
   int      getDivisor() { return denom; }
};
```

```
void Subject::notify()             // 5. Publisher broadcasts
{
   for (int i=0; i < views.size(); i++)
                      views[i]->update();
}

class DivObserver : public Observer
{
 public:
   DivObserver( Subject* mod, int div ) : Observer(mod,div) { }
   void update()                    // 6. "Pull" information of interest
   {
      int v = getSubject()->getVal(), d = getDivisor();
      cout << v << " div " << d << " is " << v / d << '\n';
   }
};

class ModObserver : public Observer
{
public:
   ModObserver( Subject* mod, int div ) : Observer(mod,div) { }
   void update()
    {
      int v = getSubject()->getVal(), d = getDivisor();
      cout << v << " mod " << d << " is " << v % d << '\n';
    }
};

void main( void )
{
   Subject      subj;
   DivObserver  divObs1( &subj,4 );  // 7. Client configures the number and
   DivObserver  divObs2( &subj,3 );  //    type of Observers
   ModObserver  modObs3( &subj,3 );
   subj.setVal( 14 );
}

// 14 div 4 is 3
// 14 div 3 is 4
// 14 mod 3 is 2
```

Example 2:

```
// Purpose.  Observer design pattern, class inheritance vs type inheritance

// SensorSystem is the "subject".  Lighting, Gates, and Surveillance are the
// "views".  The subject is only coupled to the "abstraction" of AlarmListener.
// An object's class defines how the object is implemented.  In contrast, an
// object's type only refers to its interface.  Class inheritance defines an
// object's implementation in terms of another object's implementation.  Type
// inheritance describes when an object can be used in place of another.
// [GoF, pp13-17]

#include <iostream>
#include <vector>
```

```
using namespace std;

class AlarmListener { public: virtual void alarm() = 0; };

class SensorSystem
{
  vector<AlarmListener*> listeners;
public:
  void attach( AlarmListener* al ) { listeners.push_back( al ); }
  void soundTheAlarm()
        {
            for (int i=0; i < listeners.size(); i++)
                      listeners[i]->alarm();
        }
};

class Lighting : public AlarmListener
{
  public:
        /*virtual*/ void alarm() { cout << "lights up" << '\n'; }
};

class Gates : public AlarmListener
{
  public:
        /*virtual*/ void alarm() { cout << "gates close" << '\n'; }
};

class CheckList
{
  virtual void localize() { cout << "   establish a perimeter" << '\n'; }
  virtual void isolate()  { cout << "   isolate the grid"      << '\n'; }
  virtual void identify() { cout << "   identify the source"   << '\n'; }
public:
  void byTheNumbers()           // Template Method design pattern
  {
    localize();
    isolate();
    identify();
  }
};
 // class inheri.                    // type inheritance
class Surveillance : public CheckList, public AlarmListener
{
  /*virtual*/ void isolate() { cout << "   train the cameras" << '\n'; }
public:
  /*virtual*/ void alarm()
   {
    cout << "Surveillance - by the numbers:" << '\n';
    byTheNumbers();
   }
};
```

```
void main( void )
{
   SensorSystem ss;
   ss.attach( &Gates()        );
   ss.attach( &Lighting()     );
   ss.attach( &Surveillance() );
   ss.soundTheAlarm();
}

// gates close
// lights up
// Surveillance - by the numbers:
//    establish a perimeter
//    train the cameras
//    identify the source
```

Example 3:

```
// Purpose.  Observer and Mediator demo
// Observer - 1. Sender is coupled to a Receiver interface
//            2. Receivers register with Sender
//            3. Sender broadcasts (in the blind) to all Receivers
//
// Mediator - 4. Sender(s) has handle to Mediator
//            5. Mediator has handle to Receiver(s)
//            6. Sender(s) sends to Mediator
//            7. Mediator sends to Receiver(s)

#include <iostream>
#include <vector>
using namespace std;
void gotoxy( int, int );

class Observer { public: virtual void update( int ) = 0; };

class Mediator
{
   vector<Observer*> groups[3];                    // 1. Sender is coupled to an interface
public:
   enum Message { ODD, EVEN, THREE };    // 1. Sender is coupled to an interface
   void attach( Observer* o, Message type ) { groups[type].push_back( o ); }
   void disseminate( int num )
     {
         if (num % 2 == 1)  // odd                 // 3,7. Sender/Mediator broadcasts
            for (int i=0; i < groups[0].size(); i++)
                     groups[0][i]->update(num);
         else              // even
            for (int i=0; i < groups[1].size(); i++)
                     groups[1][i]->update(num);
         if (num % 3 == 0)          // /3
            for (int i=0; i < groups[2].size(); i++)
                     groups[2][i]->update(num);
     }
};
```

```
class OddObserver : public Observer
{
   int col, row;
public:
   OddObserver( Mediator& med, int c )
   {
      col = c;          row = 3;
      gotoxy( col, 1 );  cout << "Odd";
      gotoxy( col, 2 );  cout << "---";
      med.attach( this, Mediator::ODD );  // 2,5. Receivers register with Sender
   }
   void update( int num ) { gotoxy( col, row++ );  cout << num; }
};

class EvenObserver : public Observer
{
   int col, row;
public:
   EvenObserver( Mediator& med, int c )
  {
      col = c;          row = 3;
      gotoxy( col, 1 );  cout << "/2";
      gotoxy( col, 2 );  cout << "--";
      med.attach( this, Mediator::EVEN );
   }
   void update( int num ) { gotoxy( col, row++ );  cout << num; }
};

class ThreeObserver : public Observer
 {
   int col, row;
public:
   ThreeObserver( Mediator& med, int c )
   {
      col = c;          row = 3;
      gotoxy( col, 1 );  cout << "/3";
      gotoxy( col, 2 );  cout << "--";
      med.attach( this, Mediator::THREE );
   }
   void update( int num ) { gotoxy( col, row++ );  cout << num; }
};

class Publisher
{
public:          // 6. Sender sends to Mediator
   Publisher( Mediator& med )
         {
                  for (int i=1; i < 10; i++)
                           med.disseminate(i);
         }
};
```

```
void main( void )
{
   Mediator   mediator;
   OddObserver(   mediator,  1 );
   EvenObserver(  mediator, 11 );
   ThreeObserver( mediator, 21 );
   Publisher  producer( mediator );  // 4. Sender has handle to Mediator
}

// Odd    /2       /3
// ---       --       --
// 1        2        3
// 3        4        6
// 5        6        9
// 7        8
// 9
```

Example 4:

```
// Purpose.  TypedMessage - embellished Observer, decoupled messaging
// Messages inherit from TypedMessage<self>
// Message listeners inherit from many Message::Handlers
// Application tells message to publish/broadcast/notify
// Messages are the subject (receive registrations from subscribers)
// Subsystems are the observers (receive broadcast messages)
// TypedMessage accomodates everything: registration, containment, and
//   notification of observers

#include <iostream>
#include <vector>
#include <string>
using namespace std;

template <class T> class TypedMessage
{
   static vector<Handler*> registry;
public:
   class Handler
         {
            public:
               Handler()
                     {
                        TypedMessage<T>::registerHandler( this );
                     }
               virtual void handleEvent( const T* t ) = 0;
         };
void notify()
         {
               for (int i=0; i < registry.size(); i++)
                           registry.at(i)->handleEvent( (T*)this );
         }
static void registerHandler( Handler* h )
         {
                     registry.push_back( h );
         }
};
```

```
class On : public TypedMessage<On>
{
   string comment;
public:
   On( string str )   { comment = str; }
   void start() const { cout << "OnEvent.start - " << comment << '\n'; }
};

vector<TypedMessage<On>::Handler*> TypedMessage<On>::registry;

class Off : public TypedMessage<Off>
{
   string comment;
public:
   Off( string str ) { comment = str; }
   void stop() const { cout << "OffEvent.stop - " << comment << '\n'; }
};

vector<TypedMessage<Off>::Handler*> TypedMessage<Off>::registry;

class MasterConsole : public On::Handler, public Off::Handler
{
public:
   void handleEvent( const On* msg )
            {
               cout << "MasterConsole - ";  msg->start();
            }
   void handleEvent( const Off* msg )
            {
               cout << "MasterConsole - ";  msg->stop();
            }
};

class PowerMonitor : public On::Handler
{
            public:
               void handleEvent( const On* msg )
                        {
                           cout << "PowerMonitor - ";  msg->start();
                        }
};

void main( void )
{
   MasterConsole  mc;
   PowerMonitor   pm;
   On oneEvent( "lights" );  Off thrEvent( "elevators" );
   On twoEvent( "hvac" );    Off fouEvent( "frontDoor" );
   oneEvent.notify();  twoEvent.notify();
   thrEvent.notify();  fouEvent.notify();
}

// MasterConsole - OnEvent.start - lights
// PowerMonitor - OnEvent.start - lights
// MasterConsole - OnEvent.start - hvac
// PowerMonitor - OnEvent.start - hvac
// MasterConsole - OffEvent.stop - elevators
// MasterConsole - OffEvent.stop - frontDoor
```

Example 5:

```
// Purpose.  Observer design pattern lab.
//
// Problem.  Undue coupling exists between Subject and Observer.  Subject
// has the number and type of Observer objects hard-wired in its class
// declaration.  Subject knows (or unilaterally decides) what information
// each Observer instance needs, and then "pushes" that information to the
// Observer (rather than having the Observer "pull" whatever it needs) whenever a
change occurs.
//
// Assignment.
// o Add an Observer base class for Hex, Oct, and Roman
// o Remove the Observer objects from Subject, and replace them with an
//   array of 10 pointers to the Observer base class
// o Add an "attach" method to Subject so that Observer objects can register
//themselves
// o Add a "notify" method to Subject that broadcasts to all registered
//   Observers whenever a change occurs
// o Add a "getState" method to Subject for Observer objects to use
// o Call Subject's notify() from Subject's setState().
// o Add a Subject* data member to the Observer hierarchy.
// o The Observer hierarchy constructors should accept a Subject* as an
//   argument.  The derived class ctors can either pass the Subject* they
//   receive to their base class ctor, or, they can initialize the Subject*  data member
//themselves.
// o Observer objects should attach themselves to their Subject in their   constructor.
// o Rewrite the Observer "update" method so that it requests what it wants
//   from Subject instead of receiving an input argument.
// o In main(), create one of each type of Observer

#include <iostream.h>
class HexObserver
{
public:
  void update( int val )
  {
    cout << "hex: 0x" << hex << val << endl;
  }
};

class OctObserver
{
public:
  void update( int val )
  {
    cout << "oct:  0" << oct << val << endl;
  }
};
```

```
class RomanObserver
{
public:
   void update( int val );
};


class Subject
{
public:
   void setState( int val )
  {
      subjectState = val;
      hexObserver.update( subjectState );
      octObserver.update( subjectState );
      romanObserver.update( subjectState );
  }
private:
   int            subjectState;
   HexObserver    hexObserver;
   OctObserver    octObserver;
   RomanObserver  romanObserver;
};

void main( void )
{
   Subject  subj;
   int      value;

   cout << "Input integer: ";
   while (cin >> value)
         {
               subj.setState( value );
               cout << "\nInput integer: ";
         }
}

// Input integer: 31
// hex: 0x1f
// oct:  037
// roman: XXXI
//
// Input integer: 32
// hex: 0x20
// oct:  040
// roman: XXXII
//
// Input integer: 48
// hex: 0x30
// oct:  060
// roman: XLVIII
```

```
void RomanObserver::update( int val )
{
  static char* table[3][4] = {"CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"};
  cout << "roman: ";
  if (val > 3999)
        {
            cout << "****" << endl;
            return;
        }
  while (val >= 1000)
        {
            cout << "M";
            val -= 1000;
        }
  for (int multiplier = 100, index = 0; multiplier >= 1;multiplier /= 10, index++)
        {
                 if (val >= 9 * multiplier)
                {
                      cout << table[index][0];
                      val -= 9 * multiplier;
                }
                else if (val >= 5 * multiplier)
                {
                     cout << table[index][1];
                     val -= 5 * multiplier;
                }
                 else if (val >= 4 * multiplier)
                        {
                      cout << table[index][2];
                      val -= 4 * multiplier;
                        }

                while (val >= 1 * multiplier)
                {
                      cout << table[index][3];
                      val -= 1 * multiplier;
                }
       }
  cout << endl;
}
```

State

Intent

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Problem

A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state. Or, an application is characterized by large and numerous case statements that vector flow of control based on the state of the application.

Discussion

The State pattern is a solution to the problem of how to make behavior depend on state.

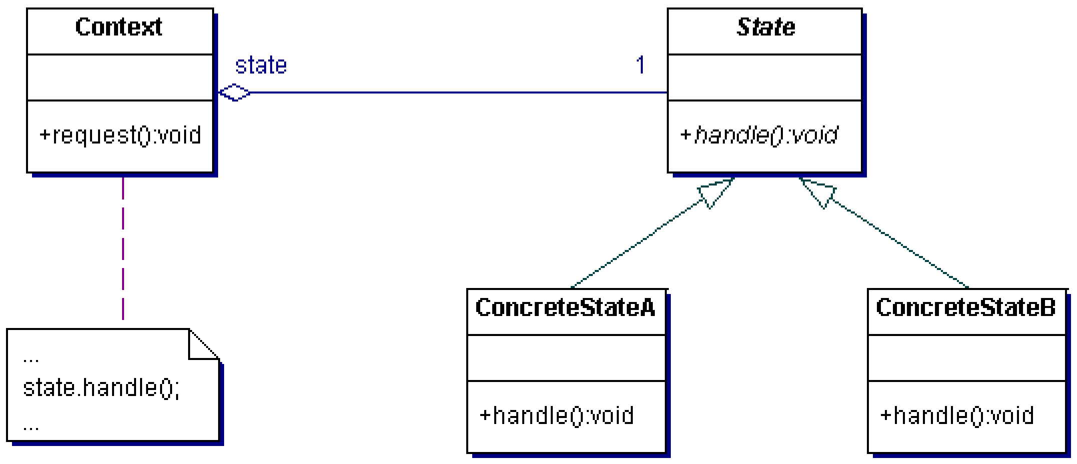
- Define a "context" class to present a single interface to the outside world.
- Define a State abstract base class.
- Represent the different "states" of the state machine as derived classes of the State base class.
- Define state-specific behavior in the appropriate State derived classes.
- Maintain a pointer to the current "state" in the "context" class.
- To change the state of the state machine, change the current "state" pointer.

The State pattern does not specify where the state transitions will be defined. The choices are two: the "context" object, or each individual State derived class. The advantage of the latter option is ease of adding new State derived classes. The disadvantage is each State derived class has knowledge of (coupling to) its siblings, which introduces dependencies between subclasses. [GOF, p308]

A table-driven approach to designing finite state machines does a good job of specifying state transitions, but it is difficult to add actions to accompany the state transitions. The pattern-based approach uses code (instead of data structures) to specify state transitions, but it does a good job of accomodating state transition actions. [GOF, p308]

The implementation of the State pattern builds on the Strategy pattern. The difference between State and Strategy is in the intent. With Strategy, the choice of algorithm is fairly stable. With State, a change in the state of the "context" object causes it to select from its "palette" of Strategy objects. [Coplien, *Multi-Paradigm Design for C*++, Addison-Wesley, 1999, p253]

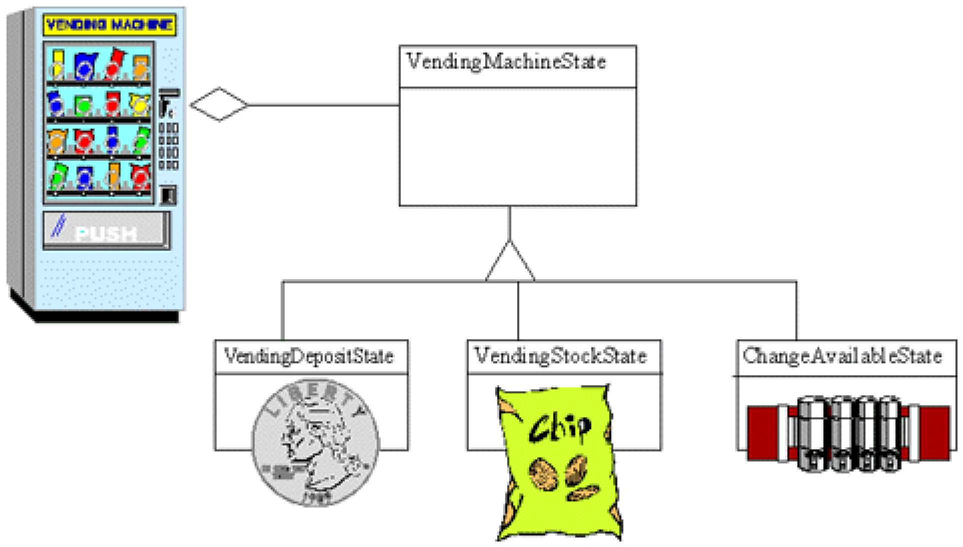
Structure



Example

The State pattern allows an object to change its behavior when its internal state changes. This pattern can be observed in a vending machine. Vending machines have states based on the inventory, amount of currency deposited, the ability to make change, the item selected, etc. When currency is deposited and a selection is made, a vending machine will either deliver a product and no change, deliver a product and change, deliver no product due to insufficient currency on deposit, or deliver no product due to inventory depletion. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]

Non-software example



Rules of thumb

State objects are often Singletons. [GOF, p313]

Flyweight explains when and how State objects can be shared. [GOF, p313]

Interpreter can use State to define parsing contexts. [GOF, p349] State is like Strategy except in its intent. [Coplien, C++ Report, Mar 96, p88] Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic). [Coplien, C++ Report, Mar 96, p88] State, Strategy, Bridge and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom [Coplien, Advanced C++, p58; see discussion under Bridge pattern]. They differ in intent - that is, they solve different problems.

The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one). The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently. [Coplien, *C++ Report*, May 95, p58]

Program Examples:

Example 1:

```
// Purpose.  State
//
// Discussion.  The boss's behavior is "morphing" radically as a function
// of his mood.  Operations have large "case" constructs that depend on
// this "state" attribute.  Like large procedures, large conditional stmts
// are undesirable.  They're monolithic, and tend to make maintenance
// very difficult.  The State pattern models individual states with de-
// rived classes of an inheritance hierarchy, and front-ends this
// hierarchy with an "interface" object that knows its "current"
// state.  This partitions and localizes all state-specific responsi-
// bilities; allowing for a cleaner implementation of dynamic behavior
// that must change as internal state changes.

class Boss
{
public:
	Boss() {	mood_ = DilbertZone; }
	void decide()
	{
		if (mood_ == DilbertZone)
		{
			cout << "Eenie, meenie,";
			cout << " mynie, moe.\n";
		}
		else if (mood_ == Sunny)
		{
			cout << "You need it - you";
			cout << " got it.\n";
		}
		toggle();
	}
	void direct()
	{
		if (mood_ == DilbertZone)
			{
				cout << "My whim - you're";
				cout << " nightmare.\n";
			}
			else if (mood_ == Sunny)
				cout << "Follow me.\n";
			toggle();
	}
private:
	enum Disposition { Sunny,DilbertZone};
	Disposition  mood_;
	void toggle() { mood_ = ! mood_; }
};
```

```
void main( void )
{
Boss ph;
for (int i=0; i < 2; i++)
        {
                ph.decide();
                ph.decide();
                ph.direct();
        }
}
```

Example 2:

```
#include <iostream.h>

class Boss
{
public:
        friend class Disposition;
        Boss();
        void decide();
        void direct();
private:
        Disposition*  moods_[2];
        int           current_;
};

class Disposition
{
public:
        virtual void decide( Boss* ) = 0;
        virtual void direct( Boss* ) = 0;
protected:
        void toggle( Boss* b )
         {
        b->current_ = ! b->current_;
        }
};

class DilbertZone :public Disposition
{
public:
        void decide( Boss* b )
        {
                cout << "Eenie, meenie, mynie,";
                cout << " moe.\n";  toggle(b);
        }
        void direct( Boss* b )
        {
                cout << "My whim - you're";
                cout << " nightmare.\n";
                toggle(b);
        }
};
```

```
class Sunny :public Disposition
{
public:
        void decide( Boss* b )
        {
                cout << "You need it - you got";
                cout << " it.\n";  toggle(b);
        }
        void direct( Boss* b )
        {
                cout << "Follow me.\n";
                toggle(b);
        }
};

        Boss::Boss()
        {
                moods_[0] = new DilbertZone;
                moods_[1] = new Sunny;
                current_ = 0;
        }
        void Boss::decide()
        {
                moods_[current_]->decide( this );
        }
        void Boss::direct()
        {
                moods_[current_]->direct( this );
        }

void main( void )
{
Boss ph;
for (int i=0; i < 2; i++)
        {
                ph.decide();
                ph.decide();
                ph.direct(); }
        }

// Eenie, meenie, mynie, moe.
// You need it - you got it.
// My whim - you're nightmare.
// You need it - you got it.
// Eenie, meenie, mynie, moe.
// Follow me.
```

Example 3:

```
// Purpose.  State design pattern - an FSM with two states and
// two events (distributed transition logic - logic in the derived state classes)
#include <iostream>              \ Event     on       off
using namespace std;       State \       -------  -------
                             ON          nothing    OFF
class Machine                OFF            ON     nothing
```

```
{
   class State* current;
public:
   Machine();
   void setCurrent( State* s ) { current = s; }
   void on();
   void off();
};

class State
{
public:
   virtual void on( Machine* m )  { cout << "   already ON\n"; }
   virtual void off( Machine* m ) { cout << "   already OFF\n"; }
};

void Machine::on()  { current->on(  this ); }
void Machine::off() { current->off( this ); }

class ON : public State
{
public:
   ON()  { cout << "   ON-ctor ";  };
   ~ON() { cout << "   dtor-ON\n"; };
   void off( Machine* m );
};

class OFF : public State
{
public:
   OFF()  { cout << "   OFF-ctor ";  };
   ~OFF() { cout << "   dtor-OFF\n"; };
   void on( Machine* m )
    {
      cout << "   going from OFF to ON";
      m->setCurrent( new ON() );
      delete this;
    }
};

void ON::off( Machine* m )
{
   cout << "   going from ON to OFF";
   m->setCurrent( new OFF() );
   delete this;
}

Machine::Machine() { current = new OFF();  cout << '\n'; }

void main( void )
{
   void (Machine::*ptrs[])() = { Machine::off, Machine::on };
   Machine fsm;
   int num;
while (1) {
                cout << "Enter 0/1: ";
                cin >> num;
                (fsm.*ptrs[num])();
          }
}

//    OFF-ctor
// Enter 0/1: 0
//    already OFF
// Enter 0/1: 1
//    going from OFF to ON   ON-ctor    dtor-OFF
// Enter 0/1: 1
//    already ON
// Enter 0/1: 0
//    going from ON to OFF   OFF-ctor    dtor-ON
// Enter 0/1: 1
//    going from OFF to ON   ON-ctor    dtor-OFF
// Enter 0/1: 0
//    going from ON to OFF   OFF-ctor    dtor-ON
// Enter 0/1: 0
//    already OFF
// Enter 0/1:
```

Example 4:

```
// Purpose.  State design pattern - an FSM with two states and
//two events
// (distributed transition logic - logic in the derived state
//classes)

// State\Event     Suck up money      Drive through
//
// RedLight          you're a            you're
//                    victim,             dead,
//                 maybe change       change to RED
//                   to GREEN
//
// GreenLight        you're an          you're free
//                     idiot            but you're
//                                    still a victim,
//                                     change to RED

#include <iostream>
#include <ctime>
using namespace std;

class FSM
{
   class State* current;
public:
   FSM();
   void setCurrent( State* s ) { current = s; }
   void suckUpMoney( int );
   void carDrivesThrough();
};
```

```
class State
{
   int total;
protected:
   int getTotal() { return total; }
public:
   State() { total = 0; }
   virtual void suckUpMoney( int in, FSM* fsm )
   {
      total += in;
      cout << "total is " << total << '\n';
   }
   virtual void carDrivesThrough( FSM* fsm ) = 0;
};

class GreenLight : public State
{
public:
   GreenLight() { cout << "GREEN light\n"; }
   void suckUpMoney( int in, FSM* fsm )
   {
      cout << "      You're an idiot, ";
      State::suckUpMoney( in, fsm );
   }
   void carDrivesThrough( FSM* fsm );
};

class RedLight : public State
{
public:
   RedLight() { cout << "RED light\n"; }
   void suckUpMoney( int in, FSM* fsm )
    {
      cout << "      ";
      State::suckUpMoney( in, fsm );
      if (getTotal() >= 50)
      {
         fsm->setCurrent( new GreenLight );
         delete this;
     }
  }
 void carDrivesThrough( FSM* fsm )
  {
      cout << "Sirens!!  Heat-seeking missile!!  Confiscate net worth!!\n";
      fsm->setCurrent( new RedLight );
      delete this;
  }
};

FSM::FSM()
{
   current = new RedLight();
}
void FSM::suckUpMoney( int in )
```

```
{
   current->suckUpMoney( in, this );
}
void FSM::carDrivesThrough()
{
   current->carDrivesThrough( this );
}
void GreenLight::carDrivesThrough( FSM* fsm )
{
   cout << "Good-bye sucker!!\n";
   fsm->setCurrent( new RedLight );
   delete this;
}

int getCoin()
{
   static int choices[3] = { 5, 10, 25 };
   return choices[rand() % 3];
}

void main( void )
{
   srand( time(0) );
   FSM fsm;
   int ans;
   while (true)
   {
      cout << "   Shell out (1), Drive thru (2), Exit (0): ";
      cin >> ans;
      if      (ans == 1) fsm.suckUpMoney( getCoin() );
      else if (ans == 2) fsm.carDrivesThrough();
      else break;
  }
}
```

Example 5:

```
// Purpose.  State demo (centralized transition logic - logic in the FSM)
// Discussion.  Who defines the state transitions?  The State pattern does not
// specify which participant defines the criteria for state transitions.  The
// logic can be implemented entirely in the Context (FSM).  It is generally
// more flexible and appropriate, however, to let the State subclasses them-
// selves specify their successor state and when to make the transition.  This
// requires adding an interface to the Context that lets State objects set the
// Context's current state explicitly.  A disadvantage of decentralization is
// that State subclasses will be coupled to other sibling subclasses. [GOF308]

#include <iostream.h>

class FSMstate
{
 public:
   virtual void on()  { cout << "undefined combo" << endl; }
   virtual void off() { cout << "undefined combo" << endl; }
   virtual void ack() { cout << "undefined combo" << endl; }
};
```

```
class FSM
{
public:
   FSM();
   void on()   { states[current]->on();  current = next[current][0]; }
   void off()  { states[current]->off(); current = next[current][1]; }
   void ack()  { states[current]->ack(); current = next[current][2]; }
private:
   FSMstate*  states[3];
   int        current;
   int        next[3][3];
};

class A : public FSMstate
{
public:
   void on()  { cout << "A, on ==> A" << endl; }
   void off() { cout << "A, off ==> B" << endl; }
   void ack() { cout << "A, ack ==> C" << endl; }
};

class B : public FSMstate
{
public:
   void off() { cout << "B, off ==> A" << endl; }
   void ack() { cout << "B, ack ==> C" << endl; }
};
class C : public FSMstate
{
public:
   void ack() { cout << "C, ack ==> B" << endl; }
};

FSM::FSM()
{
   states[0] = new A; states[1] = new B; states[2] = new C;
   current = 1;
   next[0][0] = 0; next[0][1] = 1; next[0][2] = 2;
   next[1][0] = 1; next[1][1] = 0; next[1][2] = 2;
   next[2][0] = 2; next[2][1] = 2; next[2][2] = 1;
}

enum     Message { On, Off, Ack };
Message  messageArray[10] = { On,Off,Off,Ack,Ack,Ack,Ack,On,Off,Off };

void main( void )
{
   FSM  fsm;
   for (int i = 0; i < 10; i++)
   {
      if (messageArray[i] == On)
                    fsm.on();

      else if (messageArray[i] == Off)
```

```
                fsm.off();

     else if (messageArray[i] == Ack)
                fsm.ack();
   }
}

// undefined combo            // B, ack ==> C
// B, off ==> A               // C, ack ==> B
// A, off ==> B               // undefined combo
// B, ack ==> C               // B, off ==> A
// C, ack ==> B               // A, off ==> BExample 6:
```

Example 6:

```
// Purpose.  State design pattern lab
//
// Problem.  A large monolithic case  drives the application.  As
// the program evolves, this case statemen may have to be duplicated in
// multiple places.  We want to migrate to an OO design, but we need to be
// careful not to encapsulate multiple distinct "behaviors" in a single
// object.
//
// FSM:  Message ==> on    off   ack
//       State A     A     B     C
//       State B           A     C
//       State C                 B
//
// Assignment.
// o Define an FSM context (wrapper, handle, interface) class
// o FSM declares a private data member of type FSMstate* to remember its "current
state"
// o FSM defines the member functions on(), off(), and ack().  Each of these
//   functions delegates to the contained object by calling its member func-
//   tion of the same name, and passing the FSM's "this" pointer. [GOF, p310, bottom]
// o FSM declares class FSMstate as friend (because FSMstate will be setting
//   the "current state" member)
// o Define an FSMstate base class
// o FSMstate defines a default implementation for all FSM messages (on, off,
//   ask).  The implementation of each will look like -
//      virtual void on( FSM* ) { cout << "undefined combo" << endl; }
// o FSMstate defines a protected member function -
//      void changeState( FSM*, FSMstate* )
//   This is used by the derived state classes to set the "current state"
//   member of the FSM.  This "indirection" is necessary because "friendship"
//   is not inherited by derived classes. [GOF, p311]
// o Declare states A, B, and C as derived classes of FSMstate
// o Design states A, B, and C as Singletons
// o Each of these classes override only those messages that they respond to
// o The bodies of the on(), off(), and ack() methods are simply the bodies
//   of the "case" clauses below, except that the state machine's state is
//   changed by calling the base class's changeState() method and passing the
//   "instance" of the desired FSMstate derived class.  [Note: state
//   transitions are defined in each individual state object.  This results
//   in coupling between the derived classes, but it also provides for
//   decentralization of intelligence or control.]
// o FSM's constructor initializes its "current state" member to state "B"
// o The body of main() reduces to "FSM fsm;" plus a minimal driver loop

#include <iostream.h>

enum State   { A, B, C };
enum Message { on, off, ack };
State        currentState;
Message      messageArray[10] = { on,off,off,ack,ack,ack,ack,on,off,off };

void main( void )
{
   currentState = B;
   for (int index=0; index < 10; index++)
    {
        if (currentState == A)
         {
            if (messageArray[index] == on)
            {
             cout << "A, on ==> A" << endl;
             currentState = A;
             }
             else if (messageArray[index] == off)
            {
             cout << "A, off ==> B" << endl;
              currentState = B;
            }
            else if (messageArray[index] == ack)
            {
                        cout << "A, ack ==> C" << endl;
                        currentState = C;
            }
          }
      else if (currentState == B)
            {
                     if (messageArray[index] == on)
                       {
                                   cout << "undefined combo" << endl;
                       }
                     else if (messageArray[index] == off)
                       {
                                   cout << "B, off ==> A" << endl;
                                   currentState = A;
                          }
                      else if (messageArray[index] == ack)
                           {
                                   cout << "B, ack ==> C" << endl;
                                   currentState = C;
                         }
            }
        else if (currentState == C)
            {
                     if (messageArray[index] == on)
                       {
```

```
                    cout << "undefined combo" << endl;
                }
              else if (messageArray[index] == off)
                {
                    cout << "undefined combo" << endl;
                 }
             else if (messageArray[index] == ack)
                {
                    cout << "C, ack ==> B" << endl;
                    currentState = B;
                }
         }
     }
}

// undefined combo
// B, off ==> A
// A, off ==> B
// B, ack ==> C
// C, ack ==> B
// B, ack ==> C
// C, ack ==> B
// undefined combo
// B, off ==> A
// A, off ==> B
```

Strategy

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

Problem

If clients have potentially generic algorithms embedded in them, it is difficult to: reuse these algorithms, exchange algorithms, decouple different layers of functionality, and vary your choice of policy at run-time. These embedded policy mechanisms routinely manifest themselves as multiple, monolithic, conditional expressions.

Discussion

Replace the many, monolithic, conditional constructs with a Strategy inheritance hierarchy and dynamic binding.

- Identify the protocol that provides the appropriate level of abstraction, control, and interchangeability for the client.
- Specify this protocol in an abstract base class.
- Move all related conditional code into their own concrete derived classes.
- Configure the original application with an instance of the Strategy hierarchy, and delegate to that "contained" object whenever the "algorithm" is required.

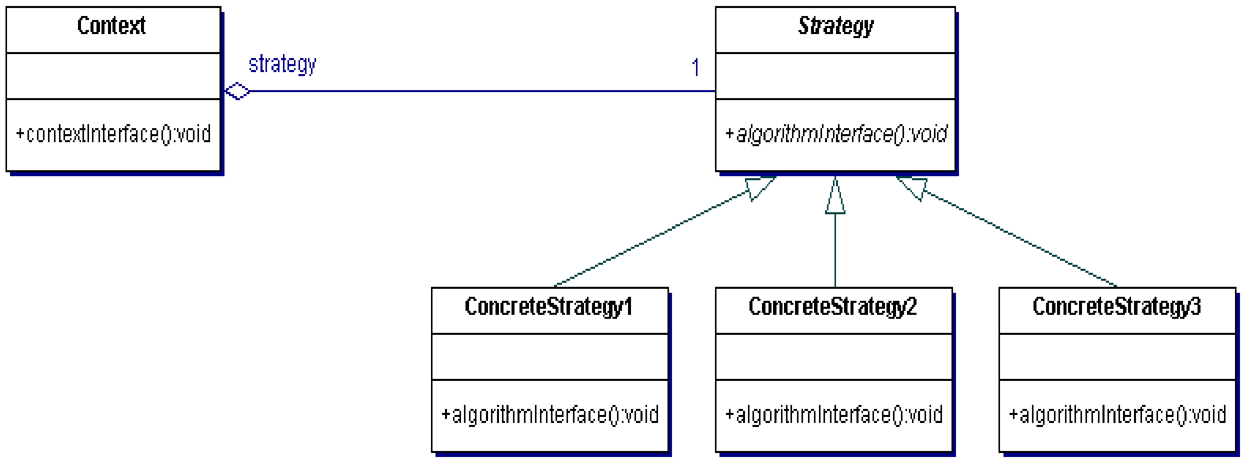
"Strategies can provide different implementations of the same behavior. The client can choose among Strategies with different time and space trade-offs." [GOF, p318] This sounds exactly like the intent of the Bridge pattern. The apparent overlap can be explained by observing that both patterns employ the same composition-delegation structure. The distinction is in intent: Strategy is a "behavioral" pattern (the focus is on delegating responsibilities), Bridge is a "structural" pattern (the focus is on establishing relationships).

The Strategy pattern should only be used when the variation in behavior is relevant to clients. If this criteria is not satisfied, the additional abstraction and effort necessary to implement the Strategy pattern is not justified.

There are two implementations for Strategy. The first is described above. It uses abstract coupling and composition-delegation. The second uses C++ templates.

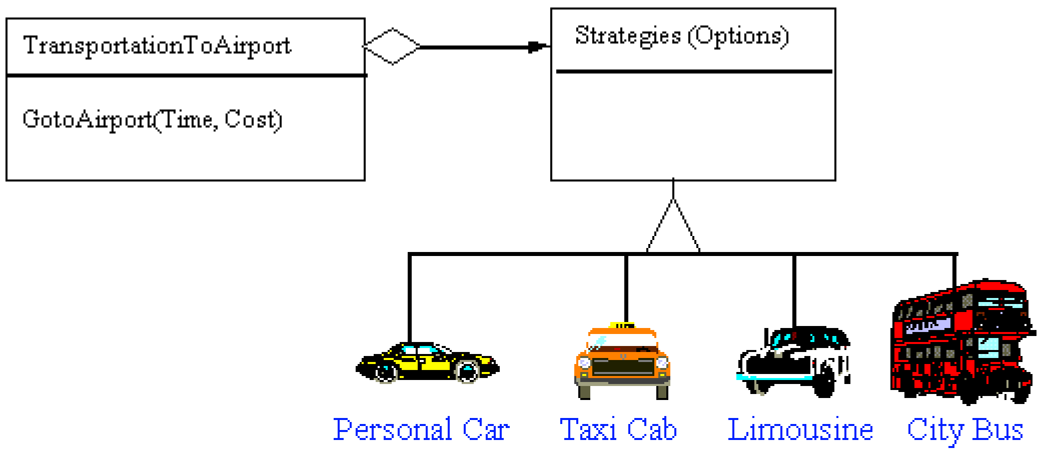
It is demonstrated in the second example that follows, and requires the client to "bind" the choice of algorithm at compile time.

Structure


Context
+contextInterface():void
strategy
1
Strategy
+algorithmInterface():void
ConcreteStrategy1
+algorithmInterface():void
ConcreteStrategy2
+algorithmInterface():void
ConcreteStrategy3
+algorithmInterface():void

Example

A Strategy defines a set of algorithms that can be used interchangeably. Modes of transportation to an airport is an example of a Strategy. Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must chose the Strategy based on tradeoffs between cost, convenience, and tIme. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

State is like Strategy except in its intent. [Coplien, Mar96, p88]

State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom [Coplien, *Advanced C++*, p58]. They differ in intent - that is, they solve different problems.

Strategy lets you change the guts of an object. Decorator lets you change the skin. [GOF, p184]

Strategy is to algorithms as Builder is to creation. [Icon, p8-13]

Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic). [Coplien, Mar96, p88]

Strategy objects often make good Flyweights. [GOF, p323]

Strategy is like Template Method except in its granularity. [Coplien, Mar96, p88]

Program Examples

Example 1:

```
// Purpose.  Strategy
//
// Discussion.  The class Stat has a Bubble sort algorithm hard-wired in
// it.  It would be nice if the choice of algorithm were configurable.
// The Strategy pattern suggests "define a family of algo's, encapsu-
// late each one, and make them interchangeable" via an abstract base class.

class Stat
 {     /* Bubble sort */
public:
          void readVector( int v[], int n )
          {
          sort_( v, n );
          min_ = v[0];   max_ = v[n-1];
          med_ = v[n/2]; }
          int getMin() { return min_; }
          int getMax() { return max_; }
          int getMed() { return med_; }
private:
          int min_, max_, med_;
          void sort_( int v[], int n )
          {
          for (int i=n-1; i > 0; i--)
                  for (int j=0; j < i; j++)
                          if (v[j] > v[j+1])
                          {
                                  int t = v[j];
                                  v[j] = v[j+1];
                                  v[j+1] = t;
                          }
          cout << "Bubble: ";
          for (int k=0; k < n; k++)
                  cout << v[k] << ' ';
          cout << endl;
          }
};

void main( void )
{
const int NUM = 9;
int     array[NUM];
time_t   t;
srand((unsigned) time(&t));
cout << "Vector: ";
for (int i=0; i < NUM; i++) {
array[i] = rand() % 9 + 1;
cout << array[i] << ' '; }
cout << endl;

Stat  obj;
obj.readVector( array, NUM );
cout << "min is " << obj.getMin()<< ", max is " << obj.getMax()<< ", median is " <<
                                                obj.getMed()<< endl;
}

/***** current implementation *****/
// Vector: 6 9 9 8 6 5 7 9 2
// Bubble: 2 5 6 6 7 8 9 9 9
// min is 2, max is 9, median is 7
/*** an upgraded implementation ***/
// Vector: 4 8 6 4 6 7 4 7 2
// Shell:  2 4 4 4 6 6 7 7 8
// min is 2, max is 8, median is 6
```

Example 2:

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>

class SortImp;
class Stat
{
public:
        Stat();
        void upGrade();
        void downGrade();
        void readVector( int[], int );
        int getMin() { return min_; }
        int getMax() { return max_; }
        int getMed() { return med_; }
private:
        int min_, max_, med_;
        SortImp*  imp_;
};

class SortImp
{
public:
        virtual void sort( int[], int ) = 0;
};

class SortBubble : public SortImp
{
public:
        void sort( int v[], int n );
};

class SortShell : public SortImp
{
public:
        void sort( int v[], int n );
};

#include "strategy2.inc"

Stat::Stat() { imp_ = new SortBubble; }
void Stat::upGrade()
{
        delete imp_;
        imp_ = new SortShell;
}

void Stat::downGrade()
{
        delete imp_;
        imp_ = new SortBubble;
}

void Stat::readVector(int v[], int n)
        {
                imp_->sort( v, n );
                min_ = v[0];   max_ = v[n-1];
                med_ = v[n/2];
        }
```

```
void main( void )
{
const int NUM = 9;
int       array[NUM];
time_t    t;
srand((unsigned) time(&t));
cout << "Vector: ";
for (int i=0; i < NUM; i++)
{
        array[i] = rand() % 9 + 1;
        cout << array[i] << ' ';
}
cout << endl;

Stat  obj;
obj.upGrade();
obj.readVector( array, NUM );
cout << "min is " << obj.getMin()<< ", max is " << obj.getMax()<< ", median is " <<
obj.getMed()<< endl;
}
```

Example 3:

```
// Purpose.  Strategy (template approach)
// A template can be used to configure a client with a Strategy.  This
// technique is appropriate if: 1) the Strategy can be selected at com-
// pile-time, and 2) it does not have to be changed at run-time.  With a
// template, there is no need to specify the interface in a SortImp
// base class.  The Stat class now has an instance of the sort object, in-
// stead of a ptr to the base class.
// The inheritance approach offers  more options and expressiveness.
// The template approach offers mildly better efficiency.

class Stat
{    /* Bubble sort */
public:
        void readVector( int v[], int n )
        {
                sort_( v, n );
                min_ = v[0];   max_ = v[n-1];
                med_ = v[n/2]; }
                int getMin() { return min_; }
                int getMax() { return max_; }
                int getMed() { return med_; }
private:
        int min_, max_, med_;
        void sort_( int v[], int n )
        {
                for (int i=n-1; i > 0; i--)
                        for (int j=0; j < i; j++)
                                if (v[j] > v[j+1])
                                {
                                        int t = v[j];
                                        v[j] = v[j+1];
                                        v[j+1] = t;
```

```
                    }
            cout << "Bubble: ";
            for (int k=0; k < n; k++)
                cout << v[k] << ' ';
            cout << endl;
        }
};

void main( void )
{
const int NUM = 9;
int     array[NUM];
time_t   t;
srand((unsigned) time(&t));
cout << "Vector: ";
for (int i=0; i < NUM; i++)
{
array[i] = rand() % 9 + 1;
cout << array[i] << ' ';
}
cout << endl;

Stat  obj;
obj.readVector( array, NUM );
cout << "min is " << obj.getMin()<< ", max is " << obj.getMax()<< ", median is " <<
obj.getMed()<< endl;
}

// Vector: 6 9 9 8 6 5 7 9 2
// Bubble: 2 5 6 6 7 8 9 9 9
// min is 2, max is 9, median is 7
```

Example 4:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

template<class STRATEGY>class Stat
{
public:
        void readVector( int v[], int n )
        {
            imp_.sort( v, n );
            min_ = v[0];   max_ = v[n-1];
            med_ = v[n/2]; }
            int getMin() { return min_;
        }
        int getMax() { return max_; }
        int getMed() { return med_; }
private:
        int     min_, max_, med_;
        STRATEGY  imp_;
};
```

```
class SortBubble
{
public:
        void sort( int v[], int n );
};

class SortShell
{
public:
        void sort( int v[], int n );
};

#include "strategy2.inc"

void main( void )
{
const int NUM = 9;
int     array[NUM];
time_t   t;
srand((unsigned) time(&t));
cout << "Vector: ";
for (int i=0; i < NUM; i++)
        {
                array[i] = rand() % 9 + 1;
                cout << array[i] << ' ';
        }
cout << endl;

Stat<SortBubble>  obj;
obj.readVector( array, NUM );
cout << "min is " << obj.getMin()<< ", max is " << obj.getMax()<< ", median is " <<
obj.getMed()<< endl;

Stat<SortShell>  two;
two.readVector( array, NUM );
cout << "min is " << two.getMin()<< ", max is " << two.getMax()<< ", median is " <<
two.getMed()<< endl;
}

// Vector: 3 5 4 9 7 1 4 9 2
// Bubble: 1 2 3 4 4 5 7 9 9
// min is 1, max is 9, median is 4
// Shell:  1 2 3 4 4 5 7 9 9
// min is 1, max is 9, median is 4
```

Example 5:

```
// Purpose.  Strategy design pattern demo
//
// Discussion.  The Strategy pattern suggests: encapsulating an algorithm
// in a class hierarchy, having clients of that algorithm hold a pointer
// to the base class of that hierarchy, and delegating all requests for
// the algorithm to that "anonymous" contained object.  In this example,
// the Strategy base class knows how to collect a paragraph of input and
// implement the skeleton of the "format" algorithm.  It defers some
// details of each individual algorithm to the "justify" member which is
// supplied by each concrete derived class of Strategy.  The TestBed class
// models an application class that would like to leverage the services of
// a run-time-specified derived "Strategy" object.

#include <iostream.h>
#include <fstream.h>
#include <string.h>

class Strategy;

class TestBed
{
public:
   enum StrategyType { Dummy, Left, Right, Center };
   TestBed()         { strategy_ = NULL; }
   void setStrategy( int type, int width );
   void doIt();
private:
   Strategy*  strategy_;
};

class Strategy
{
public:
   Strategy( int width ) : width_( width ) { }
   void format()
            {
                  char line[80], word[30];
                  ifstream  inFile( "quote.txt", ios::in );
                  line[0] = '\0';

                  inFile >> word;
                  strcat( line, word );
                  while (inFile >> word)
                  {
                     if (strlen(line) + strlen(word) + 1 > width_)
                                 justify( line );
                     else
                                 strcat( line, " " );
                     strcat( line, word );
                  }
              justify( line );
            }
protected:
   int     width_;
private:
   virtual void justify( char* line ) = 0;
};

class LeftStrategy : public Strategy
{
public:
   LeftStrategy( int width ) : Strategy( width ) { }
private:
   /* virtual */ void justify( char* line )
         {
               cout << line << endl;
               line[0] = '\0';
         }
};

class RightStrategy : public Strategy
{
public:
   RightStrategy( int width ) : Strategy( width ) { }
private:
   /* virtual */ void justify( char* line )
         {
               char  buf[80];
               int   offset = width_ - strlen( line );
               memset( buf, ' ', 80 );
               strcpy( &(buf[offset]), line );
               cout << buf << endl;
               line[0] = '\0';
         }
};

class CenterStrategy : public Strategy
{
public:
   CenterStrategy( int width ) : Strategy( width ) { }
private:
   /* virtual */ void justify( char* line )
         {
               char  buf[80];
               int   offset = (width_ - strlen( line )) / 2;
               memset( buf, ' ', 80 );
               strcpy( &(buf[offset]), line );
               cout << buf << endl;
              line[0] = '\0';
         }
};

void TestBed::setStrategy( int type, int width )
{
   delete strategy_;
   if      (type == Left)
         strategy_ = new LeftStrategy( width );
   else if (type == Right)
         strategy_ = new RightStrategy( width );
   else if (type == Center)
         strategy_ = new CenterStrategy( width );
}
void TestBed::doIt()
{
         strategy_->format();
}
```

```
void main()
{
  TestBed  test;
  int      answer, width;
  cout << "Exit(0) Left(1) Right(2) Center(3): ";
  cin >> answer;
  while (answer)
  {
    cout << "Width: ";
    cin >> width;
    test.setStrategy( answer, width );
    test.doIt();
    cout << "Exit(0) Left(1) Right(2) Center(3): ";
    cin >> answer;
  }
  return 0;
}

// Exit(0) Left(1) Right(2) Center(3): 2
// Width: 75
//       The important lesson we have learned is that development for reuse is
//  complex. If making a good design is difficult, then making a good reusable
//        design is even harder, and any amount of process description cannot
// substitute for the skill, imagination, and experience of a good designer. A
// process can only support the creative work, and ensure that things are done
//                         and recorded properly. [Karlsson et al, REBOOT]
// Exit(0) Left(1) Right(2) Center(3): 3
// Width: 75
//    The important lesson we have learned is that development for reuse is
// complex. If making a good design is difficult, then making a good reusable
//     design is even harder, and any amount of process description cannot
// substitute for the skill, imagination, and experience of a good designer. A
// process can only support the creative work, and ensure that things are done
//              and recorded properly. [Karlsson et al, REBOOT]
```

Example 6:

```
// Purpose.  Strategy design pattern lab.
//
// Problem.  Data entry validation is lumped in one function with a large
// monolithic case statement.  In a more sophisticated application there
// might be multiple parallel case statements and the validation code
// could easily become several pages in length.  If new validation
// requirements surface, the situation is exacerbated.
//
// Strategy.  "Abtract out" the validation mechanism into its own class
// hierarchy - interface in a base class, implementation in derived classes.
//
// Assignment.
// o Create a Validation base class that can serve as an umbrella for the
//   three different validation requirements
// o Move the interface of the validate() method to the Validation hierarchy
// o Move each part of the "case" statement in validate() to its own derived  class
// o In DataEntry, replace the "type" attribute and the validate() method  with a
//Validation* attribute
// o Initialize the Validation* member, inside setValidationType()
// o In interact(), delegate the validation to the Validation* member main() will stay the
//same

#include <iostream.h>
#include <string.h>

int isdigit( char ch ) { return (ch >= '0' && ch <= '9') ? 1 : 0; }
int isupper( char ch ) { return (ch >= 'A' && ch <= 'Z') ? 1 : 0; }
int islower( char ch ) { return (ch >= 'a' && ch <= 'z') ? 1 : 0; }

class DataEntry
{
public:
  void setValidationType( char ch ) { type = ch; }
  void interact();
private:
  char type;
  int  validate( char* );
};

void DataEntry::interact()
{
  char  answer[20];
  cout << "Prompt: ";
  cin >> answer;
  while (strcmp( answer, "quit" ))
  {
    if (validate( answer ))
                cout << "*** good ***" << endl;
    else
                 cout << "*** bad ***" << endl;
    cout << "Prompt: ";
    cin >> answer;
  }
}

int DataEntry::validate( char* input )
{
  int       valid;
  unsigned  i;
  valid = 1;
  if (type == 'n')
        {
            for (i=0; i < strlen(input); i++)
              if ( ! isdigit( input[i] ) )
                {
                        valid = 0;
                        break;
                }
          }
else if (type == 'u')
{
    for (i=0; i < strlen(input); i++)
              if ( ! isupper( input[i] ) )
                {
                        valid = 0;
                        break;
                }
  }
else if (type == 'l')
{
    for (i=0; i < strlen(input); i++)
              if ( ! islower( input[i] ) )
                {
                        valid = 0;
                        break;
                }
}
  return valid;
}

void main( void )
{
  DataEntry  dialog;
  char       answer;

  cout << "Input type [ (n)umber, (u)pper, (l)ower, e(x)it ]: ";
  cin >> answer;
  while (answer != 'x')
        {
            dialog.setValidationType(answer);
            dialog.interact();
            cout << "Input type [ (n)umber, (u)pper, (l)ower, e(x)it ]: ";
            cin >> answer;
        }

}

// Input type [ (n)umber, (u)pper, (l)ower, e(x)it ]: n
// Prompt: sdf
// *** bad ***
// Prompt: 234
// *** good ***
// Prompt: quit
// Input type [ (n)umber, (u)pper, (l)ower, e(x)it ]: u
// Prompt: sdf
// *** bad ***
// Prompt: 234
// *** bad ***
// Prompt: GHJ
// *** good ***
// Prompt: quit
// Input type [ (n)umber, (u)pper, (l)ower, e(x)it ]:
```

Template Method

Intent

Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Problem

Two different components have significant similarities, but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended.

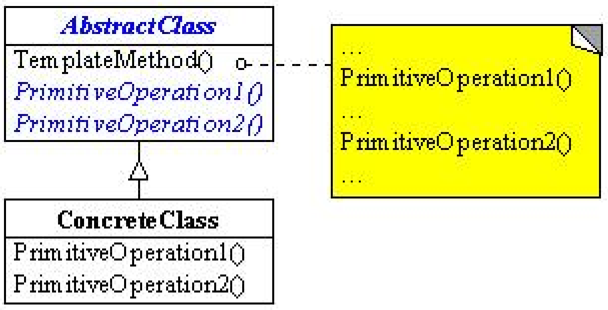
Discussion

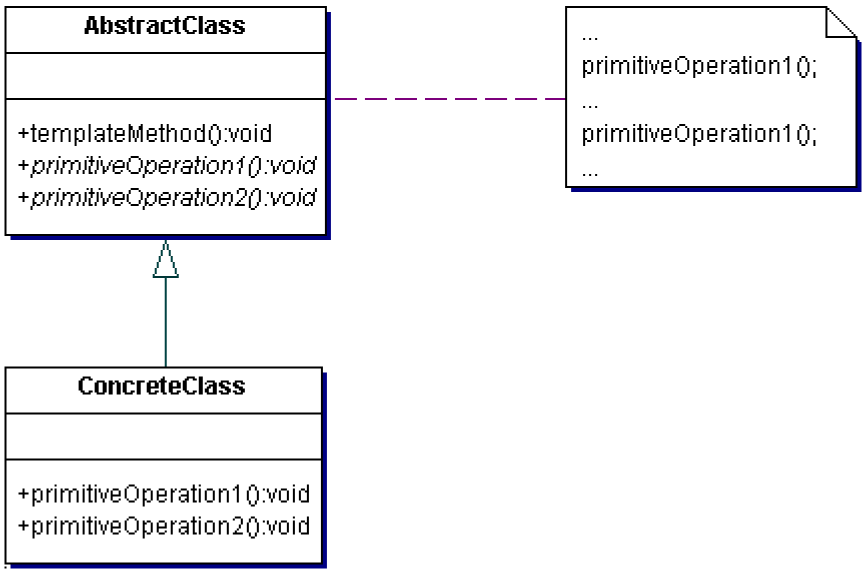
The component designer decides which steps of an algorithm are invariant (or standard), and which are variant (or customizable). The invariant steps are implemented in an abstract base class, while the variant steps are either given a default implementation, or no implementation at all. The variant steps represent "hooks", or "placeholders", that can, or must, be supplied by the component's client in a concrete derived class.

The component designer mandates the required steps of an algorithm, and the ordering of the steps, but allows the component client to extend or replace some number of these steps.

Template Method is used prominently in frameworks. Each framework implements the invariant pieces of a domain's architecture, and defines "placeholders" for all necessary or interesting client customization options. In so doing, the framework becomes the "center of the universe", and the client customizations are simply "the third rock from the sun". This inverted control structure has been affectionately labelled "the Hollywood principle" - "don't call us, we'll call you".

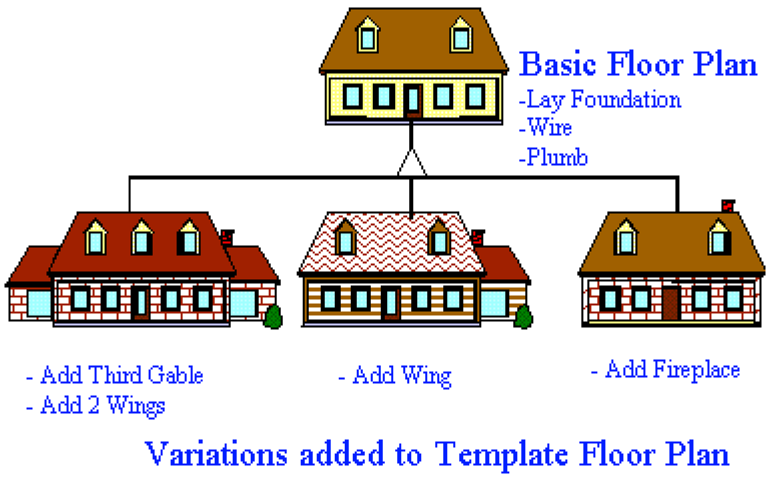
Structure




AbstractClass
+templateMethod():void
+primitiveOperation1():void
+primitiveOperation2():void
ConcreteClass
+primitiveOperation1():void
+primitiveOperation2():void
...
primitiveOperation1();
...
primitiveOperation1();
...

Example

The Template Method defines a skeleton of an algorithm in an operation, and defers some steps to subclasses. Home builders use the Template Method when developing a new subdivision. A typical subdivision consists of a limited number of floor plans with different variations available for each. Within a floor plan, the foundation, framing, plumbing, and wiring will be identical for each house. Variation is introduced in the later stages of construction to produce a wider variety of models. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54] .



Rules of thumb

Strategy is like Template Method except in its granularity. [Coplien, *C++ Report*, Mar 96, p88]

Template Method uses inheritance to vary part of an algorithm. Strategy uses delegation to vary the entire algorithm. [GOF, p330]

<u>Program Examples:</u>

Example 1:

```
// Purpose.  Template Method

#include <iostream.h>
#include <stdlib.h>
#include <time.h>

class SortUp
{  ///// Shell sort /////
public:
        void doIt( int v[], int n )
        {
        for (int g = n/2; g > 0; g /= 2)
                for (int i = g; i < n; i++)
                        for (int j = i-g; j >= 0;j -= g)
                                if (v[j] > v[j+g])
                                        doSwap(v[j],v[j+g]);
        }
private:
        void doSwap(int& a,int& b)
         {
        int t = a; a = b; b = t;
         }
};

class SortDown
{
public:
        void doIt( int v[], int n )
        {
        for (int g = n/2; g > 0; g /= 2)
                for (int i = g; i < n; i++)
                        for (int j = i-g; j >= 0;j -= g)
                                if (v[j] < v[j+g])
                                        doSwap(v[j],v[j+g]);
        }
private:
        void doSwap(int& a,int& b)
        {
        int t = a; a = b; b = t;
        }
};

void main( void )
{
const int NUM = 10;
int     array[NUM];
time_t   t;
srand((unsigned) time(&t));
for (int i=0; i < NUM; i++)
{
array[i] = rand() % 10 + 1;
cout << array[i] << ' ';
}
cout << endl;

SortUp  upObj;
upObj.doIt( array, NUM );
for (int u=0; u < NUM; u++)
        cout << array[u] << ' ';
cout << endl;

SortDown  downObj;
downObj.doIt( array, NUM );
for (int d=0; d < NUM; d++)
        cout << array[d] << ' ';
cout << endl;
}

// 3 10 5 5 5 4 2 1 5 9
// 1 2 3 4 5 5 5 5 9 10
// 10 9 5 5 5 5 4 3 2 1
```

Example 2:

```
// Purpose.  No reuse

#include <iostream>
using namespace std;

class One
{
   void a() { cout << "a  "; }
   void b() { cout << "b  "; }
   void c() { cout << "c  "; }
   void d() { cout << "d  "; }
   void e() { cout << "e  "; }
public:
   void execute() { a();  b();  c();  d();  e(); }
};

class Two
{
   void a()  { cout << "a  "; }
   void _2() { cout << "2  "; }
   void c()  { cout << "c  "; }
   void _4() { cout << "4  "; }
   void e()  { cout << "e  "; }
public:
   void execute() { a();  _2();  c();  _4();  e(); }
};

void main( void )
{
   One first;
   first.execute();
   cout << '\n';
   Two second;
   second.execute();
   cout << '\n';
}

// a  b  c  d  e
// a  2  c  4  e
```

Example 3:

```
// Purpose.  Template Method design pattern

// 1. Standardize the skeleton of an algorithm in a base class "template method"
// 2. Steps requiring peculiar implementations are "placeholders" in base class
// 3. Derived classes implement placeholder methods

#include <iostream>
using namespace std;

class Base
{
   void a() { cout << "a  "; }
   void c() { cout << "c  "; }
```

```
  void e() { cout << "e  "; }
  // 2. Steps requiring peculiar implementations are "placeholders" in base class
  virtual void ph1() = 0;
  virtual void ph2() = 0;
public:
  // 1. Standardize the skeleton of an algorithm in a base class "template method"
  void execute() {  a();  ph1();  c();  ph2();  e(); }
};

class One : public Base
{
  // 3. Derived classes implement placeholder methods
  /*virtual*/ void ph1() { cout << "b  "; }
  /*virtual*/ void ph2() { cout << "d  "; }
};

class Two : public Base
{
  /*virtual*/ void ph1() { cout << "2  "; }
  /*virtual*/ void ph2() { cout << "4  "; }
};

void main( void )
{
  Base* array[] = { &One(), &Two() };
  for (int i=0; i < 2; i++)
        {
             array[i]->execute();
             cout << '\n';
        }
}

// a  b  c  d  e
// a  2  c  4  e
```

Example 4:

```
// Purpose. Template Method design pattern

// 1. Standardize the skeleton of an algorithm in a base class "template method"
// 2. Steps requiring peculiar implementations are "placeholders" in base class
// 3. Derived classes implement placeholder methods

#include <iostream>
#include <string>
using namespace std;

class StandardAlgorithm
{
  // 3. Steps requiring peculiar implementations are "placeholders" in base class
  virtual string preprocess( char* ) = 0;
  virtual bool   validate( char )    = 0;
public:
  // 1. Standardize the skeleton of algorithm in base class "template method"
   string process( char* in )
          {
          string str = preprocess( in );
          for (int i=0; i < str.size(); i++)
               if ( ! validate( str[i] )) return "not valid";
                      return str;
         }
};

class Alphabetic : public StandardAlgorithm
{
   /*virtual*/ string preprocess( char* in )
         {
              string s( in );
              for (int i=0; i < s.size(); i++)
                 if (s[i] >= 'A' && s[i] <= 'Z' || s[i] == ' ')
                            /* empty */ ;
                 else if (s[i] >= 'a' && s[i] <= 'z')
                           s[i] = s[i] - 32;
                 else
                                 s[i] = '_';
              return s;
         }
   /*virtual*/ bool validate( char ch )
         {
              if (ch >= 'A' && ch <= 'Z' || ch == ' ')
                            return true;
             else
                            return false;
         }
};

class Numeric : public StandardAlgorithm
{
   /*virtual*/ string preprocess( char* in ) { return in; }
   /*virtual*/ bool validate( char ch )
         {
              if (ch >= '0' && ch <= '9')
                            return true;
              else
                            return false;
         }
};

void main( void )
{
   StandardAlgorithm* types[] = { &Alphabetic(), &Numeric() };
   char buf[20];
   while (true)
         {
              cout << "Input: ";
              cin.getline( buf, 20 );
              if ( ! strcmp( buf, "quit" ))
                            break;
```

```
            for (int i=0; i < 2; i++)
                cout << "   " << types[i]->process( buf ) << '\n';
        }
}

// Input: Hello World
//    HELLO WORLD
//    not valid
// Input: 12345
//    not valid
//    12345
// Input: 4.2e3
//    not valid
//    not valid
// Input: quit
```

Example 5:

```
// Purpose. Template Method design pattern

// 1. Standardize the skeleton of an algorithm in a base class "template" method
// 2. Common implementations of individual steps are defined in the base class
// 3. Steps requiring peculiar implementations are "placeholders" in base class
// 4. Derived classes can override placeholder methods
// 5. Derived classes can override implemented methods
// 6. Derived classes can override and "call back to" base class methods

#include <iostream>
using namespace std;

class A
{
public:
  // 1. Standardize the skeleton of an algorithm in a "template" method
  void findSolution()
        {
            stepOne();
            stepTwo();
            stepThr();
            stepFor();
        }
protected:
  virtual void stepFor() { cout << "A.stepFor" << '\n'; }
private:
  // 2. Common implementations of individual steps are defined in base class
  void stepOne() { cout << "A.stepOne" << '\n'; }
  // 3. Steps requiring peculiar impls are "placeholders" in the base class
  virtual void stepTwo() = 0;
  virtual void stepThr() = 0;
};

class B : public A
{
  // 4. Derived classes can override placeholder methods
  // 1. Standardize the skeleton of an algorithm in a "template" method
  /*virtual*/ void stepThr()
        {
            step3_1();
            step3_2();
            step3_3();
        }
  // 2. Common implementations of individual steps are defined in base class
  void step3_1() { cout << "B.step3_1" << '\n'; }
  // 3. Steps requiring peculiar impls are "placeholders" in the base class
  virtual void step3_2() = 0;
  void step3_3() { cout << "B.step3_3" << '\n'; }
};

class C : public B
{
  // 4. Derived classes can override placeholder methods
  /*virtual*/ void stepTwo() { cout << "C.stepTwo" << '\n'; }
  void step3_2() { cout << "C.step3_2" << '\n'; }
  // 5. Derived classes can override implemented methods
  // 6. Derived classes can override and "call back to" base class methods
  /*virtual*/ void stepFor()
        {
            cout << "C.stepFor" << '\n';
            A::stepFor();
        }
};

void main( void )
{
  C algorithm;
  algorithm.findSolution();
}

// A.stepOne
// C.stepTwo
// B.step3_1
// C.step3_2
// B.step3_3
// C.stepFor
// A.stepFor
```

Example 6:

```
// Purpose.  Template Method design pattern demo.
//
// Discussion.  The "template method" establishes the steps to be
// performed.  All standard, or invariant, steps have their implementation
// provided by the abstract base class.  All variable steps are not
// defined in the base class, but must be defined by concrete derived
// classes.  "stepFour" below is an embellishment on the design pattern
// where the base class provides a default implementation, and then the
// derived class may extend that method by: overriding the method,
// "calling-back" to the base class to leverage its implementation, and
// then adding its own peculiar behavior.
```

```
#include <iostream.h>

class IncompleteAlgorithm
{
public:
  void doIt()    // this is the Template Method
        {
            stepOne();      // invariant, standard
            stepTwo();      // invariant, standard
            stepThree();     // variable,  supplied by subclass
            stepFour();
        }    // variable,  default provided
private:
  void stepOne()
        {
         cout << "IncompleteAlgorithm::stepOne" << endl;
        }
  void stepTwo()
        {
            cout << "IncompleteAlgorithm::stepTwo" << endl;
        }
  virtual void stepThree() = 0;
protected:
  virtual void stepFour()
        {
            cout << "IncompleteAlgorithm::stepFour" << endl;
        }
};

class FillInTheTemplate : public IncompleteAlgorithm
{
  /* virtual */ void stepThree()
        {
            cout << "FillInTheTemplate::stepThree" << endl;
        }
  /* virtual */ void stepFour()
        {
            IncompleteAlgorithm::stepFour();
            cout << "FillInTheTemplate::stepFour" << endl;
        }
};

void main()
{
  FillInTheTemplate  theThingToDo;
  theThingToDo.doIt();
}

// IncompleteAlgorithm::stepOne
// IncompleteAlgorithm::stepTwo
// FillInTheTemplate::stepThree
// IncompleteAlgorithm::stepFour
// FillInTheTemplate::stepFour
// Purpose.  Template Method design pattern demo
//
// romanNumeral ::= {thousands} {hundreds} {tens} {ones}
// thousands, hundreds, tens, ones ::= nine | four | {five} {one} {one} {one}
// nine ::= "CM" | "XC" | "IX"
// four ::= "CD" | "XL" | "IV"
// five ::= 'D' | 'L' | 'V'
// one  ::= 'M' | 'C' | 'X' | 'I'
```

Example 7:

```
#include <iostream>
#include <string>
using namespace std;
class Thousand;  class Hundred;  class Ten;  class One;

class RNInterpreter
{
public:
  static int interpret( string input );
  void interpret( string& input, int& total )                // Template Method
        {
         int index = 0;
         if (input.substr(0,2) == nine())
        {
              total += 9 * multiplier();
              index += 2;
        }
        else if (input.substr(0,2) == four())
        {
              total += 4 * multiplier();
              index += 2;
          }
        else
        {
                if (input[0] == five())
                {
                        total += 5 * multiplier();
                        index = 1;
                }
              for (int end = index + 3 ; index < end; index++)
                        if (input[index] == one())
                                  total += 1 * multiplier();
                        else break;
        }
 // remove all leading chars processed
        input.replace( 0, index, "" );
       }
private:
  virtual char one()       = 0;   virtual string four() = 0;  // placeholders
  virtual char five()      = 0;   virtual string nine() = 0;  // placeholders
  virtual int  multiplier() = 0;                              // placeholders
  static Thousand thousands; static Hundred hundreds;  static Ten tens; static One
ones;
};

class Thousand : public RNInterpreter
{
  char  one()       { return 'M'; }
  string four() { return ""; }
  char  five()      { return '\0'; }
  string nine() { return ""; }
  int   multiplier() { return 1000; }
};

class Hundred : public RNInterpreter
{
  char  one()       { return 'C'; }
  string four() { return "CD"; }
  char  five()      { return 'D'; }
  string nine() { return "CM"; }
  int   multiplier() { return 100; }
};

class Ten : public RNInterpreter
{
  char  one()       { return 'X'; }   string four() { return "XL"; }
  char  five()      { return 'L'; }   string nine() { return "XC"; }
  int   multiplier() { return 10; }
};

class One : public RNInterpreter
{
  char  one()       { return 'I'; }   string four() { return "IV"; }
  char  five()      { return 'V'; }   string nine() { return "IX"; }
  int   multiplier() { return 1; }
};

Thousand RNInterpreter::thousands;
Hundred  RNInterpreter::hundreds;
Ten      RNInterpreter::tens;
One      RNInterpreter::ones;

/*static*/ int RNInterpreter::interpret( string input )
{
  int total = 0;
  thousands.interpret( input, total );
  hundreds.interpret( input, total );
  tens.interpret( input, total );
  ones.interpret( input, total );
  // if any input remains, the input was invalid, return 0
  if (input != "") return 0;
  return total;
}
```

```
void main( void )
{
  string  input;
  cout << "Enter Roman Numeral: ";
  while (cin >> input)
        {
             cout << "   interpretation is " << RNInterpreter::interpret( input ) << endl;
             cout << "Enter Roman Numeral: ";
        }
}

// Enter Roman Numeral: MCMXCVI
//    interpretation is 1996
// Enter Roman Numeral: MMMCMXCIX
//    interpretation is 3999
// Enter Roman Numeral: MMMM
//    interpretation is 0
// Enter Roman Numeral: MDCLXVIIII
//    interpretation is 0
// Enter Roman Numeral: CXCX
//    interpretation is 0
// Enter Roman Numeral: MDCLXVI
//    interpretation is 1666
// Enter Roman Numeral: DCCCLXXXVIII
//    interpretation is 888
```

Example 8:

```
// Purpose.  Template Method design pattern lab
//
// Assignment.
// o Use the Template Method pattern to refactor DirectAccessFile and
//   IndexSequentialFile into a hierarchy that demonstrates reuse of
//   implementation and interface.

#include <iostream>
using namespace std;

class DirectAccessFile
{
  void findDirectoryEntry() {cout << "find directory entry\n"; }
  void getFileTable() {cout << "get cylinder/track table\n"; }
  void computeTrack() {cout << "transform the key into its corresponding track\n"; }
  void seekToTrack() {cout << "seek to track\n"; }
  void findRecord()
  {
    cout << "read first sector\n";
    cout << "if key not found, then search track linearly to find key\n";
  }

  void readRecord() { cout << "read record\n"; }
  void bufferOutput() { cout << "buffer output\n"; }
public:
  void access()
  {
    findDirectoryEntry();
    getFileTable();
    computeTrack();
    seekToTrack();
    findRecord();
    readRecord();
    bufferOutput();
   }
};

class IndexSequentialFile
{
  void findDirectoryEntry() {cout << "find directory entry\n"; }
  void getFileTable() {cout << "get cylinder/track table\n"; }
  void searchForTrack() {cout << "take the key and search for the track\n"; }
  void seekToTrack() {cout << "seek to track\n"; }
  void searchForKey() {cout << "search track linearly to find key\n"; }
  void readRecord() {cout << "read record\n"; }
  void bufferOutput() {cout << "buffer output\n"; }
public:
  void access()
  {
    findDirectoryEntry();
    getFileTable();
    searchForTrack();
    seekToTrack();
    searchForKey();
    readRecord();
    bufferOutput();
   }
};

void main( void )
{
  DirectAccessFile    daf;
  daf.access();
  cout <<
"====================================================\n";
  IndexSequentialFile isf;
  isf.access();
}

// find directory entry
// get cylinder/track table
// transform the key into its corresponding track
// seek to track
// read first sector
// if key not found, then search track linearly to find key
// read record
// buffer output
// ========================================================
// find directory entry
// get cylinder/track table
// take the key and search for the track
// seek to track
// search track linearly to find key
// read record
// buffer output
```

Example 9:

```
// Purpose.  Template Method design pattern answer

#include <iostream>
using namespace std;

class File
{
  void findDirectoryEntry() {cout << "find directory entry\n"; }
  void getFileTable() {cout << "get cylinder/track table\n"; }
  virtual void identifyTrack() = 0;
  void seekToTrack() {cout << "seek to track\n"; }
  void readRecord() {cout << "read record\n"; }
  void bufferOutput() {cout << "buffer output\n"; }
protected:
  virtual void findRecord() { cout << "search track linearly to find key\n"; }
public:
  void access()
  {
    findDirectoryEntry();
    getFileTable();
    identifyTrack();
    seekToTrack();
    findRecord();
    readRecord();
    bufferOutput();
  }
};

class DirectAccessFile : public File
{
  /*virtual*/ void identifyTrack()
   {
    cout << "transform the key into its corresponding track\n";
   }
  /*virtual*/ void findRecord()
   {
    cout << "read first sector\n";
    cout << "if key not found, then ";
    File::findRecord();
   }
};
```

```
class IndexSequentialFile : public File
{
   /*virtual*/ void identifyTrack()
   {
      cout << "take the key and search for the track\n";
   }
};

void main( void )
{
   File* files[] = { &DirectAccessFile(), &IndexSequentialFile() };
   files[0]->access();
   cout <<
"============================================================\n";
   files[1]->access();
}

// find directory entry
// get cylinder/track table
// transform the key into its corresponding track
// seek to track
// read first sector
// if key not found, then search track linearly to find key
// read record
// buffer output
// ========================================================
// find directory entry
// get cylinder/track table
// take the key and search for the track
// seek to track
// search track linearly to find key
// read record
// buffer output
```

Visitor

Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Problem

Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure. You want to avoid "polluting" the node classes with these operations. And, you don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.

Discussion

Visitor's primary purpose is to abstract functionality that can be applied to an aggregate hierarchy of "element" objects. The approach encourages designing lightweight Element classes - because processing functionality is removed from their list of responsibilities. New functionality can easily be added to the original inheritance hierarchy by creating a new Visitor subclass.

Visitor implements "double dispatch". OO messages routinely manifest "single dispatch" - the operation that is executed depends on: the name of the request, and the type of the receiver. In "double dispatch", the operation executed depends on: the name of the request, and the type of TWO receivers (the type of the Visitor and the type of the element it visits).

The implementation proceeds as follows. Create a Visitor class hierarchy that defines a pure virtual visit() method in the abstract base class for each concrete derived class in the aggregate node hierarchy. Each visit() method accepts a single argument - a pointer or reference to an original Element derived class.

Each operation to be supported is modelled with a concrete derived class of the Visitor hierarchy. The visit() methods declared in the Visitor base class are now defined in each derived subclass by allocating the "type query and cast" code in the original implementation to the appropriate overloaded visit() method.

Add a single pure virtual accept() method to the base class of the Element hierarchy. accept() is defined to receive a single argument - a pointer or reference to the abstract base class of the Visitor hierarchy.

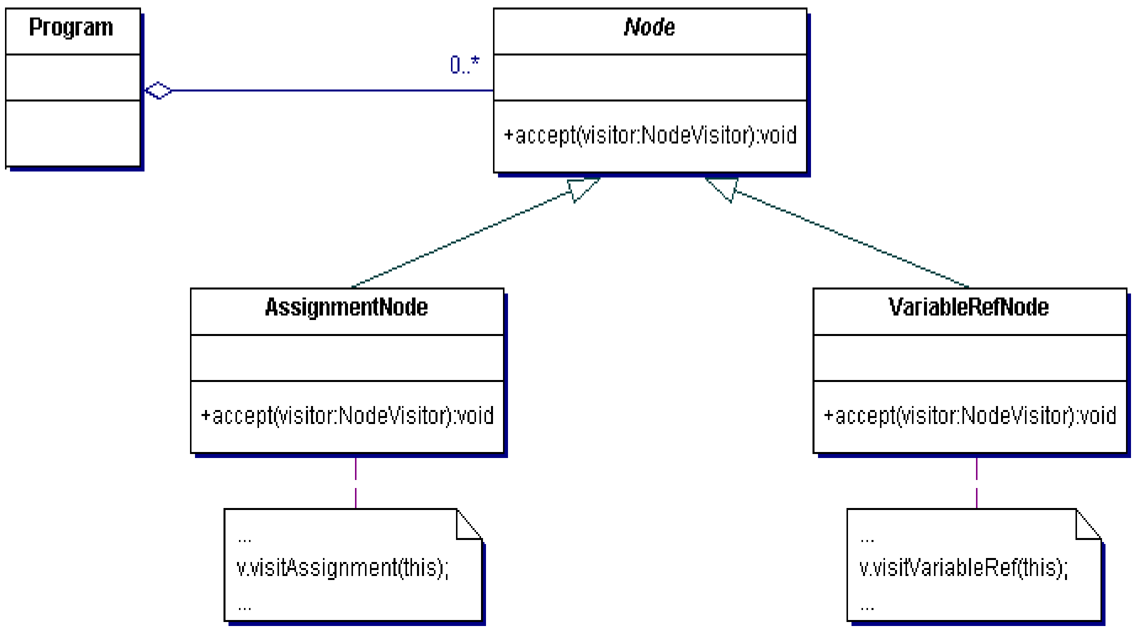
Each concrete derived class of the Element hierarchy implements the accept() method by simply calling the visit() method on the concrete derived instance of the Visitor hierarchy that it was passed, passing its "this" pointer as the sole argument.

Everything for "elements" and "visitors" is now set-up. When the client needs an operation to be performed, (s)he creates an instance of the Vistor object, calls the accept() method on each Element object, and passes the Visitor object. The accept() method causes flow of control to find the correct Element subclass. Then when the visit() method is invoked, flow of control is vectored to the correct Visitor subclass. accept() dispatch plus visit() dispatch equals double dispatch.

The Visitor pattern makes adding new operations (or utilities) easy - simply add a new Visitor derived class. But, if the subclasses in the aggregate node hierarchy are not stable, keeping the Visitor subclasses in sync requires a prohibitive amount of effort.

An acknowledged objection to the Visitor pattern is that is represents a regression to functional decomposition - separate the algorithms from the data structures. While this is a legitimate interpretation, perhaps a better perspective/rationale is the goal of promoting non-traditional behavior to full object status.

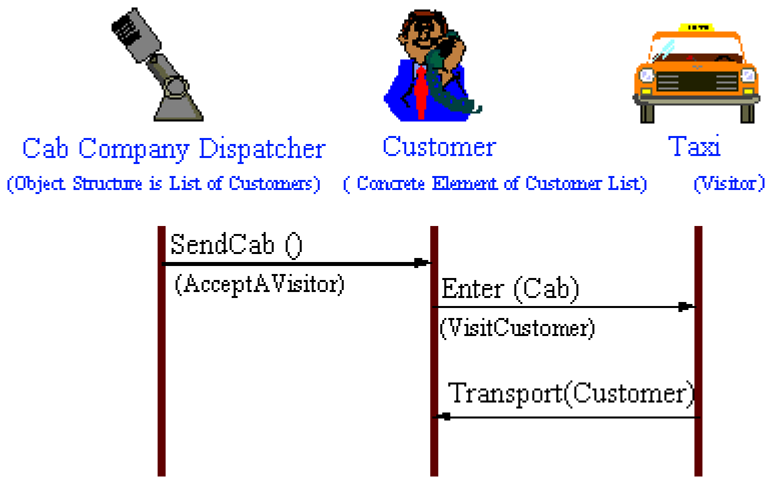
Structure


Program
Node
0..*
+accept(visitor:NodeVisitor):void
AssignmentNode
+accept(visitor:NodeVisitor):void
VariableRefNode
+accept(visitor:NodeVisitor):void
...
v.visitAssignment(this);
...
...
v.visitVariableRef(this);
...

The *NodeVisitor* hierarchy is not modeled here. The first dispatch is *accept()*, and the second dispatch is *visit()*. Inside each *visit()* method, we now know the exact type of both the *Node* entity (the object being passed) and the *NodeVisitor* entity (the object being messaged).

Example

The Visitor pattern represents an operation to be performed on the elements of an object structure without changing the classes on which it operates. This pattern can be observed in the operation of a taxi company. When a person calls a taxi company (accepting a visitor), the company dispatches a cab to the customer. Upon entering the taxi the customer, or Visitor, is no longer in control of his or her own transportation, the taxi (driver) is. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable). [GOF, p255]

Iterator can traverse a Composite. Visitor can apply an operation over a Composite. [GOF, p173]

The Visitor pattern is like a more powerful Command pattern because the visitor may initiate whatever is appropriate for the kind of object it encounters. [Johnson, Huni, Engel, 1995, p8]

The Visitor pattern is the classic technique for recovering lost type information without resorting to dynamic casts. [Vlissides, "Type Laundering", *C++ Report*, Feb 97, p48]

Program Examples:

Example 1:

```
// Purpose.  Visitor (adding ops)Discussion.  On the left, adding
// new operations requires modifyingall the current Color classes.  If
// we introduce an extra level of indirection (a Visitor hierarchy),
// then operations capable of operating on Color classes can be added
// without modifying any existing code.  On the right, an entry point
// for all future operations has been added in the form of the accept()
// method.  Each of the previous operations has been encapsulated in
// its own derived Visitor class.  Any new operations simply require the
// addition of a new Visitor class.
//
// When we call accept() on a Color object, dynamic binding gets us to
// the correct derived class of Color.
// Then when we call visit() on the Visitor object, dynamic binding
// gets us to the correct derived class of Visitor.  [Visitors and
// Colors can be passed by address or passed by reference.

#include <iostream.h>

class Color
{
public:
	virtual void count() = 0;
	virtual void call()  = 0;
	static void reportNum()
	{
	    cout << "Reds " << numRed_ <<", Blus " << numBlu_ <<endl;
	}
protected:
	static int numRed_, numBlu_;
};

int Color::numRed_ = 0;
int Color::numBlu_ = 0;

class Red : public Color
{
public:
	void count() { numRed_++; }
	void call() { eye(); }
	void eye() { cout << "Red::eye\n";}
};

class Blu : public Color
{
public:
	void count() { numBlu_++; }
	void call() { sky(); }
	void sky() { cout << "Blu::sky\n";}
};
```

```
void main( void )
{
Color* set[] = { new Red, new Blu,new Blu, new Red, new Red, 0 };
for (int i=0; set[i]; i++)
  {
    set[i]->count();
    set[i]->call();
  }
Color::reportNum();
}
```

Example 2:

```
#include <iostream.h>
class Visitor;

class Color
{
public:
        virtual void accept( Visitor& ) = 0;
};

class Red : public Color
{
public:
        void accept( Visitor& );
        void eye() { cout << "Red::eye\n"; }
};

class Blu : public Color
{
public:
        void accept( Visitor& );
        void sky() { cout << "Blu::sky\n"; }
};

class Visitor
{
public:
        virtual void visit( Red& ) = 0;
        virtual void visit( Blu& ) = 0;
};

class CountV : public Visitor
{
public:
        CountV() { numRed_ = numBlu_ = 0; }
        virtual void visit( Red& ) { numRed_++; }
        virtual void visit( Blu& ) {numBlu_++; }
        void reportNum()
        {
                cout << "Reds " << numRed_ <<", Blus " << numBlu_ << endl;
        }
private:
        int  numRed_, numBlu_;
};

class CallV : public Visitor
{
public:
        virtual void visit( Red& r ) {r.eye(); }
        virtual void visit( Blu& b ) { b.sky(); }
};

void Red::accept( Visitor& v ) { v.visit( *this ); }
void Blu::accept( Visitor& v ) {v.visit( *this ); }

void main( void )
{
Color* set[] = { new Red, new Blu,new Blu, new Red, new Red, 0 };
CountV  countOp;
CallV   callOp;
for (int i=0; set[i]; i++)
        {
        set[i]->accept( countOp );
        set[i]->accept( callOp );
        }
countOp.reportNum();
}

// Red::eye
// Blu::sky
// Blu::sky
// Red::eye
// Red::eye
// Reds 3, Blus 2
```

Example 3:

```
// Purpose.  Visitor (double dispatch)
//
// Discussion.  On the left, the State derived classes must query the type of
// the Cmd objects they receive, in order to identify what the next course of
// action is.  "case" stmts are always a maintenance headache.  On the right,
// we have recognized that what we really want to do is "dispatch" based on
// the type of TWO objects, (a State object and a Cmd object).  The call to
// accept() discriminates the type of the State object that is being messaged,
// and then the call to visit() discriminates the type of the Cmd object
// (while passing the type of the State object).  If new Cmd classes are
// added, no change whatsoever is necessary in the code of the State classes.
// If new State classes are added, then every Cmd class must be changed, and
// Visitor is NOT the right approach to take.

#include <iostream.h>
int  current = 0;
enum CmdTyp { OnT, OffT };
```

```
class Cmd
{
public:
        virtual CmdTyp typ() = 0;
};

class On : public Cmd
{
public:
        CmdTyp typ() { return OnT; }
};

class Off : public Cmd
{
public:
        CmdTyp typ() { return OffT; }
};

class State
{
public:
        virtual void process( Cmd* c ) {cout << "ERROR\n"; }
};

class One : public State
{
public:

        void process( Cmd* c )
        {
                if (c->typ() == OnT)
                {
                current = 1;
                cout << "One,On => Two\n";
                }
                else if (c->typ() == OffT)
                        State::process( c );
        }
};

class Two : public State
{
public:
        void process( Cmd* c )
        {
                if (c->typ() == OnT)
                        State::process( c );
                else if (c->typ() == OffT)
                {
                        current = 0;
                        cout << "Two,Off => One\n";
                }
        }
};
```

```
State* states[] = { new One, new Two };

void main( void )
{
Cmd* c[] = { new Off,new On, new Off, new Off, 0 };
for (int i=0; c[i]; i++)
        states[current]->process( c[i] );
}

// ERROR
// One,On => Two
// Two,Off => One
// ERROR
```

Example 4:

```
#include <iostream.h>
int  current = 0;
class One;  class Two;

class Cmd
{
public:
        virtual void visit( One* )
        {
                cout << "ERROR\n";
        }
        virtual void visit( Two* )
        {
        cout << "ERROR\n";
        }
};

class On : public Cmd
{
public:
        void visit( One* )
        {
        current = 1;
        cout << "One,On => Two\n";
        }
        void visit( Two* t )
        {
        Cmd::visit( t );
        }
};

class Off : public Cmd
{
public:
        void visit( One* o )
        {
        Cmd::visit( o );
        }
        void visit( Two* )
        {
        current = 0;
        cout << "Two,Off => One\n";
        }
};

class State
{
public:
        virtual void accept( Cmd* c ) = 0;
};

class One : public State
{
public:
        void accept( Cmd* c )
        {
        c->visit( this );
        }
};

class Two : public State
{
public:
        void accept( Cmd* c )
        {
        c->visit( this );
        }
};

State* states[] = { new One, new Two };

void main( void )
{
Cmd* c[] = { new Off,new On, new Off, new Off, 0 };
for (int i=0; c[i]; i++)
        states[current]->accept( c[i] );
}

// ERROR
// One,On => Two
// Two,Off => One
// ERROR
```

Example 5:

```
// Purpose.  Visitor design pattern

// 1. Add an accept(Visitor) method to the "element" hierarchy
// 2. Create a "visitor" base class w/ a visit() method for every "element" type
// 3. Create a "visitor" derived class for each "operation" to do on "elements"
// 4. Client creates "visitor" objects and passes each to accept() calls

#include <iostream>
#include <string>
```

```
using namespace std;

// 1. Add an accept(Visitor) method to the "element" hierarchy
class Element
{
public:
  virtual void accept( class Visitor& v ) = 0;
};

class This : public Element
{
public:
  /*virtual*/ void accept( Visitor& v );
  string thiss() { return "This"; }
};

class That : public Element
{
public:
  /*virtual*/ void accept( Visitor& v );
  string that() { return "That"; }
};

class TheOther : public Element
{
public:
  /*virtual*/ void accept( Visitor& v );
  string theOther() { return "TheOther"; }
};

// 2. Create a "visitor" base class w/ a visit() method for every "element" type
class Visitor
{
public:
  virtual void visit( This* e ) = 0;
  virtual void visit( That* e ) = 0;
  virtual void visit( TheOther* e ) = 0;
};

/*virtual*/ void This::accept( Visitor& v )     { v.visit( this ); }
/*virtual*/ void That::accept( Visitor& v )     { v.visit( this ); }
/*virtual*/ void TheOther::accept( Visitor& v ) { v.visit( this ); }

// 3. Create a "visitor" derived class for each "operation" to do on "elements"
class UpVisitor : public Visitor
{
  /*virtual*/ void visit( This* e ) { cout << "do Up on " + e->thiss() << '\n'; }
  /*virtual*/ void visit( That* e ) { cout << "do Up on " + e->that() << '\n'; }
  /*virtual*/ void visit( TheOther* e ) {cout << "do Up on " + e->theOther() << '\n'; }
};

class DownVisitor : public Visitor
{
/*virtual*/ void visit( This* e ) { cout << "do Down on " + e->thiss() << '\n'; }
/*virtual*/ void visit( That* e ) { cout << "do Down on " + e->that() << '\n'; }
/*virtual*/ void visit( TheOther* e ) { cout << "do Down on " + e->theOther() << '\n'; }
};

void main( void )
{
   Element* list[] = { new This(), new That(), new TheOther() };
   UpVisitor    up;             // 4. Client creates
   DownVisitor  down;           //    "visitor" objects
   for (int i=0; i < 3; i++)    //    and passes each
      list[i]->accept( up );    //    to accept() calls
   for (i=0; i < 3; i++)
      list[i]->accept( down );
}

// do Up on This                // do Down on This
// do Up on That                // do Down on That
// do Up on TheOther            // do Down on TheOther
```

Example 6:

```
// Purpose.  Visitor - recovering lost type information
//
// Motivation.  "My Component classes do not know that Composites exist.
// They provide no help for navigating Composites, nor any help for
// altering the contents of a Composite.  This is because I would like the
// base class (and all its derivatives) to be reusable in contexts that do
// not require Composites.  When given a base class pointer, if I
// absolutely need to know whether or not it is a Composite, I will use
// dynamic_cast() to figure this out.  In those cases where dynamic_cast()
// is too expensive, I will use a Visitor." [Robert Martin]

#include <iostream>
#include <vector>
using namespace std;

class Visitor
{
public:
   virtual void visit( class Primitive*, class Component* ) = 0;
   virtual void visit( class Composite*, Component* ) = 0;
};

class Component
{
   int value;
public:
   Component( int val )    { value = val; }
   virtual void traverse() { cout << value << " "; }
   // Having add() here sacrifices safety, but it supports transparency
   // virtual void add( Component* ) { }
   virtual void accept( Visitor&, Component* ) = 0;
};
```

```
class Primitive : public Component
{
public:
   Primitive( int val ) : Component( val ) { }
   /*virtual*/ void accept( Visitor& v, Component* c ) { v.visit( this, c ); }
};

class Composite : public Component
{
   vector<Component*> children;
public:
   Composite( int val ) : Component( val ) { }
   void add( Component* ele ) { children.push_back( ele ); }
   /*virtual*/ void accept( Visitor& v, Component* c ) { v.visit( this, c ); }
   /*virtual*/ void traverse()
     {
      Component::traverse();
      for (int i=0; i < children.size(); i++)
         children[i]->traverse();
     }
};

class AddVisitor : public Visitor
{
public:
   /*virtual*/ void visit( Primitive*, Component* ) {/* does not make sense */}
   /*virtual*/ void visit( Composite* node, Component* c ) { node->add( c ); }
};

void main( void )
{
   Component*  nodes[3];
   // The type of Composite* is "lost" when the object is assigned to a
   // Component*
   nodes[0] = new Composite(1);
   nodes[1] = new Composite(2);
   nodes[2] = new Composite(3);

   // If add() were in class Component, this would work
   //    nodes[0]->add( nodes[1] );
   // If it is NOT in Component, and only in Composite,  you get the error -
   //    no member function `Component::add(Component *)' defined

   // Instead of sacrificing safety, we use a Visitor to support add()
   AddVisitor  addVisitor;
   nodes[0]->accept( addVisitor, nodes[1] );
   nodes[0]->accept( addVisitor, nodes[2] );
   nodes[0]->accept( addVisitor, new Primitive(4) );
   nodes[1]->accept( addVisitor, new Primitive(5) );
   nodes[1]->accept( addVisitor, new Primitive(6) );
   nodes[2]->accept( addVisitor, new Primitive(7) );
```

```
  for (int i=0; i < 3; i++)
  {
    nodes[i]->traverse();
    cout << endl;
  }
}

// 1 2 5 6 3 7 4
// 2 5 6
// 3 7
```

Example 7:

```
// Purpose.  Combining Visitor with Composite's recursive traversal

#include <iostream>
#include <vector>
using namespace std;

class Visitor
{
public:
  virtual void visit( class Leaf* e )      = 0;
  virtual void visit( class Composite* e ) = 0;
};

class Component
{
public:
  virtual void traverse() = 0;
  virtual void accept( class Visitor& v ) = 0;
};

class Leaf : public Component
{
  int value;
public:
  Leaf( int val ) { value = val; }
  /*virtual*/ void traverse() { cout << value << ' '; }
  /*virtual*/ void accept( class Visitor& v ) { v.visit( this ); }
  int getValue() { return value; }
};

class Composite : public Component
{
  char value;
  vector<Component*> children;
  static char next;
public:
  Composite() { value = next++; }
  void add( Component* ele ) { children.push_back( ele ); }
  /*virtual*/ void traverse()
   {
     cout << value << ' ';
     for (int i=0; i < children.size(); i++)
       children[i]->traverse();
   }

  /*virtual*/ void accept( class Visitor& v )
   {
     v.visit( this );
     // accept() has been embellished to include the logic in traverse()
     for (int i=0; i < children.size(); i++)
       children[i]->accept( v );
   }

  char getValue() { return value; }
};
char Composite::next = 'a';

class TransformVisitor : public Visitor
{
public:
  /*virtual*/ void visit( Leaf* e ) { cout << e->getValue() + 100 << ' '; }
  /*virtual*/ void visit( Composite* e ) { cout << (char)(e->getValue()-32) <<' ';}
};

void main( void )
{
  Composite containers[4];
  for (int i=0; i < 4; i++)
     for (int j=0; j < 3; j++)
        containers[i].add( new Leaf( i * 3 + j ) );
  for (i=1; i < 4; i++)
          containers[0].add( &(containers[i]) );

 containers[0].traverse();
 cout << endl;

  TransformVisitor tv;
  // don't need an "iteration" capability with this design
  containers[0].accept( tv );
  cout << endl;
}

// a 0 1 2 b 3 4 5 c 6 7 8 d 9 10 11
// A 100 101 102 B 103 104 105 C 106 107 108 D 109 110 111
```

Example 8:

```
// Purpose.  Undesireable design - double dispatch - doing the right thing based
// on the type of two objects

#include <iostream>
#include <string>
using namespace std;

class Request { public: virtual string getType() = 0; };

class R1 : public Request
{
public:
  /*virtual*/ string getType() { return "One"; }
  void reqOneMethod( class P1* );
  void reqOneMethod( class P2* );
};

class R2 : public Request
{
public:
  /*virtual*/ string getType() { return "Two"; }
  void reqTwoMethod( P1* );
  void reqTwoMethod( P2* );
};

class Processor { public: virtual void handle( class Request* ) = 0; };

class P1 : public Processor
{
public:
  /*virtual*/ void handle( class Request* req )
  {
    if (req->getType() == string("One"))
                ((R1*)req)->reqOneMethod( this );
    else if (req->getType() == string("Two"))
                ((R2*)req)->reqTwoMethod( this );
  }
  void procOneMethod() { cout << "processor one handling "; }
};

class P2 : public Processor
{
public:
  /*virtual*/ void handle( class Request* req )
   {
    if (req->getType() == string("One"))
                ((R1*)req)->reqOneMethod( this );
    else if (req->getType() == string("Two"))
                ((R2*)req)->reqTwoMethod( this );
   }
   void procTwoMethod() { cout << "processor two handling "; }
};

void R1::reqOneMethod( P1* p ) { p->procOneMethod(); cout << "request one\n"; }
void R1::reqOneMethod( P2* p ) { p->procTwoMethod(); cout << "request one\n"; }
void R2::reqTwoMethod( P1* p ) { p->procOneMethod(); cout << "request two\n"; }
void R2::reqTwoMethod( P2* p ) { p->procTwoMethod(); cout << "request two\n"; }

void main( void )
{
   Processor* handlers[] = { new P1(), new P2() };
   Request*   commands[] = { new R1(), new R2() };
   for (int i=0; i < 2; i++)
      for (int j=0; j < 2; j++)
         handlers[i]->handle( commands[j] );
}

// processor one handling request one
// processor one handling request two
// processor two handling request one
// processor two handling request two
```

Example 9:

```
// Purpose.  Visitor - double dispatch - doing the right thing based on the
// type of two objects

#include <iostream>
#include <string>
using namespace std;

class Request
{
public:
   // second dispatch - the "visit()" method
   virtual void execute( class P1* ) = 0;
   virtual void execute( class P2* ) = 0;
};

class R1 : public Request
{
public:
   /*virtual*/ void execute( P1* );
   /*virtual*/ void execute( P2* );
};

class R2 : public Request
{
public:
   /*virtual*/ void execute( P1* );
   /*virtual*/ void execute( P2* );
};
```

```
class Processor
{
public:
   // first dispatch - the "accept()" method
   virtual void handle( class Request* ) = 0;
};
class P1 : public Processor
{
public:
   /*virtual*/ void handle( Request* req ) { req->execute( this ); }
   void procOneMethod() { cout << "processor one handling "; }
};

class P2 : public Processor
{
public:
   /*virtual*/ void handle( Request* req ) { req->execute( this ); }
   void procTwoMethod() { cout << "processor two handling "; }
};

/*virtual*/ void R1::execute( P1* p ) { p->procOneMethod(); cout << "request one\n"; }
/*virtual*/ void R1::execute( P2* p ) { p->procTwoMethod(); cout << "request one\n"; }
/*virtual*/ void R2::execute( P1* p ) { p->procOneMethod(); cout << "request two\n"; }
/*virtual*/ void R2::execute( P2* p ) { p->procTwoMethod(); cout << "request two\n"; }

void main( void )
{
   Processor* handlers[] = { new P1(), new P2() };
   Request*   commands[] = { new R1(), new R2() };
   for (int i=0; i < 2; i++)
      for (int j=0; j < 2; j++)
         handlers[i]->handle( commands[j] );
}

// processor one handling request one
// processor one handling request two
// processor two handling request one
// processor two handling request two
```

Example 10:

```
// Purpose.  Acyclic Visitor design pattern [PLOPD vol 3, p93]
//
// Problem.  In GOF Visitor, Element depends on Visitor, Visitor depends on
// all Element derivatives, and Element derivatives depend on Element; this is
// cyclic dependency.  Additionally, adding an Element derivative requires the
// entire Visitor hierarchy to change.  "These problems can be solved by using
// multiple inheritance and dynamic_cast()."
//
// Solution.  Element derived classes are only coupled to Visitor base class.
// Visitor derived classes are only coupled to the Element derived classes that
// they choose to be coupled to.  If a new Element derived class is added,
// Visitor derived classes can update themselves if, and when, they choose.
```

```
#include <iostream>
#include <string>
using namespace std;

class Element
{
public:
  virtual void accept( class Visitor& v ) = 0;
};

class This : public Element
{
public:
  /*virtual*/ void accept( Visitor& v );
  string thiss() { return "This"; }
};

class That : public Element
{
public:
  /*virtual*/ void accept( Visitor& v );
  string that() { return "That"; }
};

class TheOther : public Element
{
public:
  /*virtual*/ void accept( Visitor& v );
  string theOther() { return "TheOther"; }
};

class Visitor
{
public:
  virtual ~Visitor() { };
};

class ThisVisitor
{
public:
  virtual void visit( This* e ) = 0;
};

class ThatVisitor
{
public:
  virtual void visit( That* e ) = 0;
};

class TheOtherVisitor
{
public:
  virtual void visit( TheOther* e ) = 0;
};
```

```
/*virtual*/ void This::accept( Visitor& v )
{
   ThisVisitor* tv = dynamic_cast<ThisVisitor*>( &v );
   if (tv) tv->visit( this );
   else cout << "the visitor was not accepted\n";
}

/*virtual*/ void That::accept( Visitor& v )
{
   ThatVisitor* tv = dynamic_cast<ThatVisitor*>( &v );
   if (tv) tv->visit( this );
   else cout << "the visitor was not accepted\n";
}

/*virtual*/ void TheOther::accept( Visitor& v )
{
   TheOtherVisitor* tv = dynamic_cast<TheOtherVisitor*>( &v );
   if (tv) tv->visit( this );
   else cout << "the visitor was not accepted\n";
}

class UpVisitor : public Visitor, public ThisVisitor, public ThatVisitor,
                                                         public TheOtherVisitor
{
   /*virtual*/ void visit( This* e ) {cout << "do Up on " + e->thiss() << '\n'; }
   /*virtual*/ void visit( That* e ) {cout << "do Up on " + e->that() << '\n'; }
   /*virtual*/ void visit( TheOther* e ) {cout << "do Up on " + e->theOther() << '\n'; }
};

class DownVisitor : public Visitor, public ThisVisitor, public ThatVisitor,
                                                          public TheOtherVisitor
 {
   /*virtual*/ void visit( This* e ) {cout << "do Down on " + e->thiss() << '\n'; }
   /*virtual*/ void visit( That* e ) {cout << "do Down on " + e->that() << '\n'; }
   /*virtual*/ void visit( TheOther* e ) {cout << "do Down on " + e->theOther() << '\n'; }
};

void main( void )
{
   Element* list[] = { new This(), new That(), new TheOther() };
   UpVisitor    up;            // 4. Client creates
   DownVisitor  down;          //    "visitor" objects
   for (int i=0; i < 3; i++)   //    and passes each
      list[i]->accept( up );   //    to accept() calls
   for (i=0; i < 3; i++)
      list[i]->accept( down );
}

// do Up on This              // do Down on This
// do Up on That              // do Down on That
// do Up on TheOther          // do Down on TheOther
```

Example 10:

```
// Purpose.  Double dispatch (within a single hierarchy)
//
// Discussion.  We would like to declare a function like:
//    void process( virtual Base* object1, virtual Base* object2 )
// that does the right thing based on the type of 2 objects that come from
// a single inheritance hierarchy.  The only problem is that the keyword
// "virtual" may not be used to request dynamic binding for an object being
// passed as an argument.  C++ will only "discriminate" the type of an object
// being messaged, not the type of an object being passed.  So in order for
// the type of 2 objects to be discriminated, each object must be the
// receiver of a virtual function call.  Here, when process1() is called on
// the first object, its type becomes "known" at runtime, but the type of
// the second is still UNknown.  process2() is then called on the second
// object, and the identity (and type) of the first object is passed as an
// argument.  Flow of control has now been vectored to the spot where the
// type (and identity) of both objects are known.

#include <iostream>
using namespace std;

class Base
{
public:
   virtual void process1( Base& ) = 0;
   virtual void process2( class A& ) = 0;
   virtual void process2( class B& ) = 0;
   virtual void process2( class C& ) = 0;
};

class A : public Base
{
public:
   /*virtual*/ void process1( Base& second ) { second.process2( *this ); }
   /*virtual*/ void process2( class A& first ) {cout << "first is A, second is A\n"; }
   /*virtual*/ void process2( class B& first ) {cout << "first is B, second is A\n"; }
   /*virtual*/ void process2( class C& first ) {cout << "first is C, second is A\n"; }
};

class B : public Base
{
public:
   /*virtual*/ void process1( Base& second ) { second.process2( *this ); }
   /*virtual*/ void process2( class A& first ) {cout << "first is A, second is B\n"; }
   /*virtual*/ void process2( class B& first ) {cout << "first is B, second is B\n"; }
   /*virtual*/ void process2( class C& first ) {cout << "first is C, second is B\n"; }
};

class C : public Base
{
public:
   /*virtual*/ void process1( Base& second ) { second.process2( *this ); }
   /*virtual*/ void process2( class A& first ) {cout << "first is A, second is C\n"; }
   /*virtual*/ void process2( class B& first ) {cout << "first is B, second is C\n"; }
   /*virtual*/ void process2( class C& first ) {cout << "first is C, second is C\n"; }
};

void main( void )
{
   Base* array[] = { &A(), &B(), &C() };
   for (int i=0; i < 3; i++)
      for (int j=0; j < 3; j++)
         array[i]->process1( *array[j] );
}

// first is A, second is A
// first is A, second is B
// first is A, second is C
// first is B, second is A
// first is B, second is B
// first is B, second is C
// first is C, second is A
// first is C, second is B
// first is C, second is C
```

Example 11.

```
// Purpose.  Triple dispatch (within a single hierarchy)
//
// Discussion.  It would be nice if C++ supported creating a function like:
// "processCombine( virtual Binary& first, virtual Binary& second, virtual
// Binary& third )".  While not directly supported, we can simulate this kind
// of capability by: calling combine() on the first object to resolve its
// type, then calling combine() on the second object to resolve its type,
// then calling combine() on the third object to resolve its type.  We
// "remember" the type information we have "discriminated" at each stage by
// juggling the three objects through 2 increasingly type-specific parameter
// slots.

#include <iostream>
using namespace std;

class Zero;
class One;

class Binary
{
public:
   // First dispatch
   virtual void combine( Binary& second, Binary& third ) = 0;
   // Second dispatch
   virtual void combine( Binary& third, Zero& first ) = 0;
   virtual void combine( Binary& third, One&  first ) = 0;
   // Third dispatch
   virtual void combine( Zero& first, Zero& second ) = 0;
   virtual void combine( Zero& first, One&  second ) = 0;
   virtual void combine( One& first,  Zero& second ) = 0;
   virtual void combine( One& first,  One&  second ) = 0;
};
```

```
class Zero : public Binary
{
public:
   void combine( Binary& second, Binary& third ) {second.combine( third, *this ); }
   void combine( Binary& third, Zero& first ) {third.combine( first, *this ); }
   void combine( Binary& third, One& first ) {third.combine( first, *this ); }

   void combine( Zero& first, Zero& second );
   void combine( Zero& first, One&  second );
   void combine( One&  first, Zero& second );
   void combine( One&  first, One&  second );

   void doZero() { cout << "0  "; }
};

class One : public Binary
{
public:
   void combine( Binary& second, Binary& third ) {second.combine( third, *this ); }
   void combine( Binary& third, Zero& first ) {third.combine( first, *this ); }
   void combine( Binary& third, One& first ) {third.combine( first, *this ); }

   void combine( Zero& first, Zero& second );
   void combine( Zero& first, One&  second );
   void combine( One&  first, Zero& second );
   void combine( One&  first, One&  second );

   void doOne() { cout << "1  "; }
};

void Zero::combine( Zero& first, Zero& second )
{
         first.doZero();
         second.doZero();
         doZero();
         cout << endl;
}
void Zero::combine( Zero& first, One&  second )
{
   first.doZero();
   second.doOne();
   doZero();
   cout << endl;
}

void Zero::combine( One&  first, Zero& second )
{
   first.doOne();
   second.doZero();
   doZero();
   cout << endl;
}
```

```
void Zero::combine( One&  first, One&  second )
{
   first.doOne();
   second.doOne();
   doZero();
   cout << endl;
}

void One::combine( Zero& first, Zero& second )
{
   first.doZero();
   second.doZero();
   doOne();
   cout << endl;
}

void One::combine( Zero& first, One&  second )
{
   first.doZero();
   second.doOne();
   doOne();
   cout << endl;
}

void One::combine( One&  first, Zero& second )
{
   first.doOne();
   second.doZero();
   doOne();
   cout << endl;
}

void One::combine( One&  first, One&  second )
{
   first.doOne();
   second.doOne();
   doOne();
   cout << endl;
}

void processCombine( Binary& first, Binary& second, Binary& third )
{
   first.combine( second, third );
}

void main( void )
{
   Binary*  list[2] = { &Zero(), &One() };

   // Run through permutations
   for (int i=0; i < 2; i++)
      for (int j=0; j < 2; j++)
         for (int k=0; k < 2; k++)
            processCombine( *list[i], *list[j], *list[k] );
}
```

```
// 0  0  0
// 0  0  1
// 0  1  0
// 0  1  1
// 1  0  0
// 1  0  1
// 1  1  0
// 1  1  1
```

Example 12:

```
// Purpose.  Visitor design pattern lab
// Problem.  "Open for extension, closed for modification" is a dominant
// principle of object-oriented design.  The Composite design pattern is an
// excellent strategy for designing recursively, or hierarchically, related
// collections of objects.  Once the Composite hierarchy is produced, adding
// new methods (that operate on the hierarchy) requires either: violating the
// "open-closed" principle, or, writing client code that has to do "type
// checking" and "type casting" in order to "recover lost type information".
// This latter approach is modeled below in the cat() and wc() methods of
// class Client.  The Visitor pattern specifies that a single access method
// be added to the Composite hierarchy.  Once this is done, many new methods
// can be added in the future, the "open-closed" principle can be enforced,
// and, the ugly code for "recovering lost type information" can be avoided.
// Assignment.
// o Define a Visitor base class with 3 pure virtual member functions:
//      void visit( Primitive* ) = 0;
//      void visit( Link* )      = 0;
//      void visit( Composite* ) = 0;
// o Define 2 derived classes of Visitor (CatVisitor and WcVisitor) that
//   encapsulate the ugly code in class Client.  The implementation of each of
//   the overloaded visit() methods will be the body of one of the conditional
//   clauses of the existing cat() and wc() methods.
// o Add a pure virtual member function to Component:
//      void accept( Visitor& ) = 0;
// o Define accept() in all 3 Component derived classes as:
//      void accept( Visitor& v ) { v.visit( this ); }
// o Create CatVisitor and WcVisitor objects in main().
// o The body of main()'s "for" loop will become:
//      array[i]->accept( aCatObj );
//      array[i]->accept( aWcObj );
// o Get rid of compType, returnType(), Client.

#include <iostream.h>
enum compType { PrimitiveT, LinkT, CompositeT };

class Component
{
public:
   virtual compType  returnType()       = 0;
   virtual void      streamOut()        = 0;
   virtual void      add( Component* )  { };
};
```

```
class Primitive : public Component
{
public:
   Primitive( int id )   { identity = id; }
   compType returnType() { return PrimitiveT; }
   void     streamOut()  { cout << identity << " "; }
private:
   int identity;
};

class Link : public Component
{
public:
   Link( Component& ele ) : linkElement(ele)  { }
   compType   returnType()                    { return LinkT; }
   void       streamOut()                     { linkElement.streamOut(); }
   Component& getSubject()                    { return linkElement; }
private:
   Component& linkElement;
};

class Composite : public Component
{
public:
   Composite()                   { index = 0; }
   compType returnType()         { return CompositeT; }
   void     add( Component* ele ) { array[index++] = ele; }
   void     streamOut()
   {
      for (int i=0; i < index; i++)
         array[i]->streamOut();
   }
private:
   int        index;
   Component*  array[20];
};

class Client
{
public:
   void cat( Component* node )
   {
      if (node->returnType() == PrimitiveT)
            {
         cout << "cat: ";   node->streamOut();
            cout << endl;
        }
      else if (node->returnType() == CompositeT)
         cout << "cat: Can't cat a directory." << endl;
      else if (node->returnType() == LinkT)
         cat(&(((Link*)node)->getSubject()));
   }
   void wc( Component* node )
   {
      if (node->returnType() == PrimitiveT)
       {
          cout << "wc: ";   node->streamOut();
          cout << endl;
       }
      else if (node->returnType() == CompositeT)
         cout << "wc: Can't wc a directory." << endl;
      else if (node->returnType() == LinkT)
         wc(&(((Link*)node)->getSubject()));
  }
};

void main( void )
{
   Composite        dir;
   Primitive        file1(1), file2(2);
   Link             alias( file1 );
   Component*       array[4];
   Client           doFile;

   dir.add( &file1 );
   dir.add( &file2 );
   dir.add( &alias );
   dir.streamOut();
   cout << endl;

   array[0] = &dir;
   array[1] = &file1;
   array[2] = &file2;
   array[3] = &alias;
   for (int i = 0; i < 4; i++)
   {
      doFile.cat( array[i] );
      doFile.wc( array[i] );
   }
}

// 1 2 1
// cat: Can't cat a directory.
// wc: Can't wc a directory.
// cat: 1
// wc: 1
// cat: 2
// wc: 2
// cat: 1
// wc: 1
```

Notes

The November 2000 issue of *JavaPro* has an article by James Cooper (author of a Java companion to the GoF) on the Visitor design pattern. He suggests it "turns the tables on our object-oriented model and creates an external class to act on data in other classes ... while this may seem unclean ... there are good reasons for doing it."

His primary example. Suppose you have a hierarchy of Employee-Engineer-Boss. They all enjoy a normal vacation day accrual policy, but, Bosses also participate in a "bonus" vacation day program. As a result, the interface of class Boss is different than that of class Engineer. We cannot polymorphically traverse a Composite-like organization and compute a total of the organization's remaining vacation days. "The Visitor becomes more useful when there are several classes with different interfaces and we want to encapsulate how we get data from these classes." [It also is useful when you want to perform the right algorithm, based on the type of two (or more) objects (aka "double dispatch"). Example algorithms might be: process a "collision" between different kinds of SpaceGame objects, compute the distance between different kinds of shapes, or compute the intersection between different kinds of shapes.]

His benefits for Visitor include:

- Add functions to class libraries for which you either do not have the source or cannot change the source
- Obtain data from a disparate collection of unrelated classes and use it to present the results of a global calculation to the user program
- Gather related operations into a single class rather than force you to change or derive classes to add these operations
- Collaborate with the Composite pattern

Visitor is not good for the situation where "visited" classes are not stable. Every time a new Composite hierarchy derived class is added, every Visitor derived class must be amended.

Notes from the Sep-Oct 2000 issue of *IEEE Software*. McGraw and Felten (authors of 2 Java security books) describe their Jslint tool (statically scans Java code for security vulnerabilities). The end of the article summarized - "Jslint parses the source file into a syntax tree that it can traverse using the Visitor design pattern. This pattern encapsulates each operation on a syntax tree into a single object called a

Visitor, allowing users to define new operations on a tree without changing the tree elements. It encodes each [security] rule in a single Visitor that traverses the parse tree looking for instances of the violation in question ..... Users can easily port our tool to scan other languages by following three simple steps:"

1. replace the Java grammar
2. write a new set of Visitor classes
3. add a new user interface

Who ya gonna call?

• The system shall be "platform-independent" • The system shall be independent of how its constituent pieces are created, composed, and represented. • The system shall be configured with one of multiple families of products. • A family of products designed to be used together shall have their relationship constraints enforced. • "case" statements are a maintenance nightmare - they shall be avoided.	Abstract Factory
• The algorithm for creating a complex subsystem shall be independent of the parts that make up the subsystem and how the parts are assembled. • The creation process for a complex subsystem shall support multiple representations of that subsystem.	Builder
• A subsystem shall not have the "type" of component it is responsible for creating hard-wired. • The system shall be "open for extension, but closed for modification".	Factory Method
• The overhead of component creation shall be localized and minimized. Instead of creating entirely new instances of a component type, a "prototypical" instance of each type shall be designed to "clone" itself. • Decisions about the type of component to create shall be deferred until application run-time.	Prototype
• A single instance of a subsystem shall be enforced, and the subsystem itself shall be responsible for that enforcement. • The single instance shall be globally accessible. • The single instance shall be initialized only if, and when, it is accessed.	Singleton
• A legacy component shall be reused in a new design, and the interface "impedance mismatch" shall be reconciled. • A reusable component shall cooperate with unrelated (or unforeseen) application components.	Adapter
• Component interfaces shall be decoupled from their implementation(s). • The client shall be "insulated" from implementation	Bridge

changes. [Changes to implementation details should only require a re-link, not a re-compile.] • The choice of implementation shall be configurable at run-time. • Both the "interface" and the "implementation" shall be separately "open for extension, but closed for modification". • "Interface" components that represent "implementation" components that are "expensive" and "equal" shall share these implementations, instead of duplicating them.	
• The system shall support recursive composition. • The system shall support "whole-part" hierarchical assemblies of components. • Clients shall be able to transparently interact with compositions of components (e.g. a sub-tree) and individual components (e.g. a node).	Composite
• Components shall be extensible at run-time. • Functionality shall be "layer-able". The client shall be capable of configuring any combination of capabilities by simply specifying the layers (or wrappers, or onion skins) to be applied.	Decorator
• The system shall provide a simple interface to a complex subsystem. • The system shall provide alternative novice, intermediate, and "power-user" interfaces. • The system shall decouple subsystems from each other. • The system shall support "layering" of subsystems.	Facade
• The system shall employ hundreds of components at very fine levels of granularity without prohibitive cost. • The system shall support pooling and reuse of a finite supply of "x" across an inexhaustible demand.	Flyweight
• The system shall support "distributed processing" by providing a local representative for a component in a different address space. • The system shall support "lazy creation" - a component is created only if, and when, the client demonstrates an interest in it. • The system shall control access to components by interposing an intermediary that evaluates the identity and access priviledges of the requestor. • The system architecture shall be characterized by multiple dimensions of indirection that offer a locus for intelligence that would unduly complicate other components if they were obliged to support the additional responsibility.	Proxy

• The system shall allow the client to issue a request to one of several "handlers" without knowing or specifying the receiver explicitly. • The system shall allow a suite of "handlers" to be configured at run-time. • The system shall allow the client to "launch and leave" a request with a "virtual pipeline of handlers". • "Senders" and "receivers" shall be decoupled from one another.	Chain of Responsibility
• The system architecture shall provide a "callback" framework. • The system shall encapsulate "execute" requests so that they may be created, queued, and subsequently serviced. They may also be logged, archived, loaded, and re-applied. • The system shall support hierarchical compositions of primitive "command" abstractions. • The system shall support reuse of "command" abstractions. [Define a "command", and simultaneously attach the encapsulation to a push button, a toolbar icon, a menu item, and a keyboard accelerator - so that all users of the GUI can enjoy their preferred method of interaction.] • The system shall support "redo" • The system shall support transactions • "Senders" and "receivers" shall be decoupled from one another.	Command
• The system shall characterize the domain as a set of rules, then define a language capable of specifying the rules, and a grammar for defining the language, and an "engine" for interpreting the grammar.	Interpreter
• The system shall support accessing an aggregate component's contents without exposing its internal representation. • The system shall support multiple simultaneous traversals of aggregate components without complicating the implementation of the aggregate itself. • The system shall define a uniform interface for traversing dissimilar aggregate components. • The system shall decouple "data structures" from "algorithms" so that each can be developed, maintained, and used independent of the other.	Iterator
• The system shall encapsulate complex, many-to-many coupling between "peer" components in a separate component capable of allowing the "peers" to be: disengaged, replaced, and reused. • The system shall support numerous many-to-many "mappings" to be defined, installed, and exchanged by the client.	Mediator

• The system shall balance the distribution of intelligence emphasized by "logical" OO design with the centralization of intelligence often required by "physical" large scale design. • "Senders" and "receivers" shall be decoupled from one another.	
• The system shall support "undo" or "rollback" • The system shall support transactions • The system shall support saving, or "flattening", or "streaming" components without compromising their encapsulation.	Memento
• The system shall support multiple "views" of the same "model" (or "subscribers" to the same "publisher", or "consumers" to the same "producer"). • Each "model" component shall be decoupled from the number and type of its "view" components. • Each "view" component shall be capable of driving the flow of information from the "model" to itself. • "Independent" components shall be decoupled from "dependent" components. • "Senders" and "receivers" shall be decoupled from one another.	Observer
• Components shall be capable of "morphing" their behavior at run-time. • The system architecture shall be characterized by a "finite state machine". The state machine will need to support application logic at each state transition, not simply the transition itself. [A "table-driven" approach routinely supports only the latter.] • "case" statements are a maintenance nightmare - they shall be avoided.	State
• Clients shall be decoupled from "choice of algorithm". • Clients shall be decoupled from complex, algorithm-specific data structures. • The choice, or implementation, of algorithm shall be configurable at run-time. • "case" statements are a maintenance nightmare - they shall be avoided.	Strategy
• Individual steps of an algorithm shall be defineable or replaceable. • The system shall provide "hooks" wherever future functionality or extensibility may be required. • The "invariant" parts of an algorithm shall be implemented in one place, and reused all other places.	Template Method

• The "standard" parts of an algorithm shall be implemented and enforced in one place. • Reuse shall be emphasized by architecting a "don't call us, we'll call you" framework. • The system shall be "open for extension, but closed for modification".	
• The system shall allow the operations that can be performed on "whole-part" hierarchical assemblies of components to be defined and added without having to change (and potentially break) any existing code. • The system shall decouple "data structures" from "algorithms" so that each can be developed, maintained, and used independent of the other. • The design shall support the recovery of "lost type information" without resorting to the overhead associated with RTTI. • "case" statements are a maintenance nightmare - they shall be avoided. • The system shall be "open for extension, but closed for modification".	Visitor