

|  |  |
| --- | --- |
| **Sr. No.** | **Creational** |
| **1** | **Singleton** |
| **2** | **Factory Method** |
| **3** | **Abstract Factory** |
| **4** | **Builder** |
| **5** | **Prototype** |
|  |  |
| **Sr. No.** | **Structural** |
| **1** | **Composite** |
| **2** | **Bridge** |
| **3** | **Proxy** |
| **4** | **Flyweight** |
| 5 | Adapter |
| 6 | Façade |
| 7 | Decorator |
|  |  |
| **Sr. No.** | **Behavioral** |
| **1** | **Observer** |
| **2** | **Mediator** |
| **3** | **Chain of responsibility** |
| **4** | **Iterator** |
| 5 | Command |
| 6 | Interpreter |
| 7 | Memento |
| 8 | Strategy |
| 9 | State |
| 10 | template Method |
| 11 | Visitor |

**Design Patterns:**

**Creational:**

**Abstract Factory:**

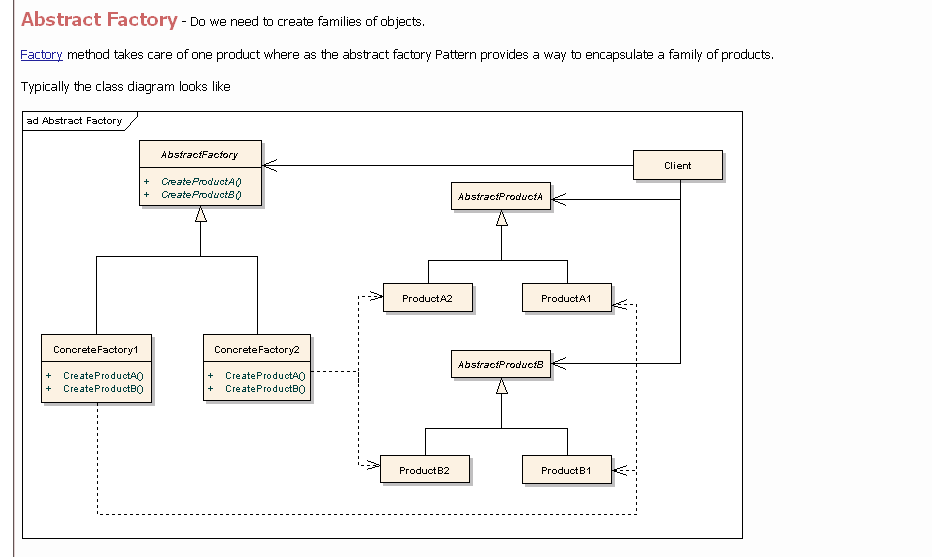
|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Abstract Factory | Kit | Provide an interface for creating families of related or dependent objects without specifying their concrete classes. |

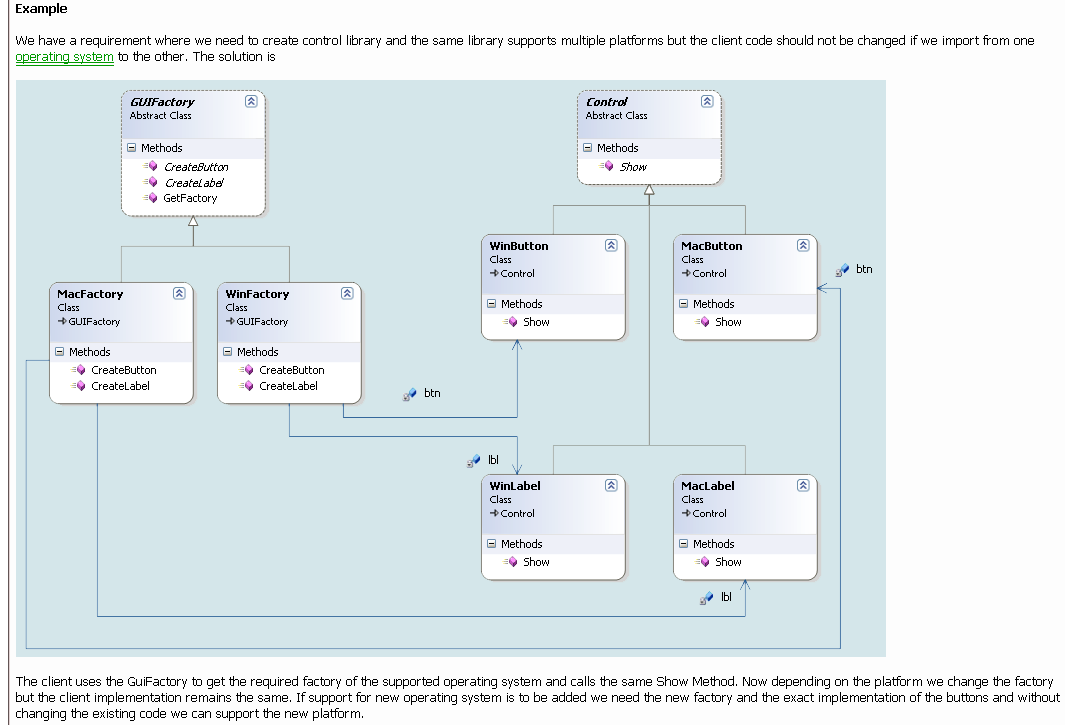
Example:

Abstract Factory for supporting multiple look-and-feel standards. In case study, …Supporting multiple look-and-feel standards. Lexi should adapt easily to different look-and-feel standards such as Motif and Presentation Manager (PM) without major modification.   
GUIFactory - MotifFactory, PMFactory, MacFactory  
Glyph - ScrollBar, Button, Menu  
ScrollBar - MotiffScrollBar, PMScrollBar, MacScrollBar.....similar for button & Menu  
Example from Doc:  
GUIFactory - MacFactory, WinFactory  
Controls - Button, Label  
Button - MacButton, WinButton  
Label - MacLabel, WinLabel

Related Patterns:

AbstractFactory classes are often implemented with factory methods (Factory Method), but they can also be implemented using Prototype.  
A concrete factory is often a singleton (Singleton).





#define WINDOWS

class Widget {

public:

virtual void draw() = 0;

};

class MotifButton : public Widget {

public:

void draw() { cout << "MotifButton\n"; }

};

class MotifMenu : public Widget {

public:

void draw() { cout << "MotifMenu\n"; }

};

class WindowsButton : public Widget {

public:

void draw() { cout << "WindowsButton\n"; }

};

class WindowsMenu : public Widget {

public:

void draw() { cout << "WindowsMenu\n"; }

};

class Factory {

public:

virtual Widget\* create\_button() = 0;

virtual Widget\* create\_menu() = 0;

};

class MotifFactory : public Factory {

public:

Widget\* create\_button() {

return new MotifButton; }

Widget\* create\_menu() {

return new MotifMenu; }

};

class WindowsFactory : public Factory {

public:

Widget\* create\_button() {

return new WindowsButton; }

Widget\* create\_menu() {

return new WindowsMenu; }

};

Factory\* factory;

void display\_window\_one() {

Widget\* w[] = { factory->create\_button(),

factory->create\_menu() };

w[0]->draw(); w[1]->draw();

}

void display\_window\_two() {

Widget\* w[] = { factory->create\_menu(),

factory->create\_button() };

w[0]->draw(); w[1]->draw();

}

int main( void ) {

#ifdef MOTIF

factory = new MotifFactory;

#else // WINDOWS

factory = new WindowsFactory;

#endif

Widget\* w = factory->create\_button();

w->draw();

display\_window\_one();

display\_window\_two();

}

// WindowsButton

// WindowsButton

// WindowsMenu

// WindowsMenu

// WindowsButton

**Builder:**

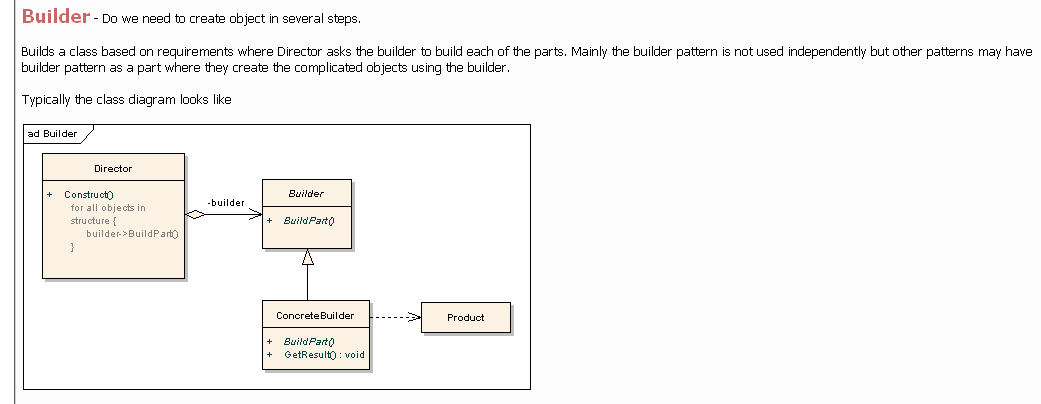
|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Builder |  | Separate the construction of a complex object from its representation so that the same construction process can create different representations. |

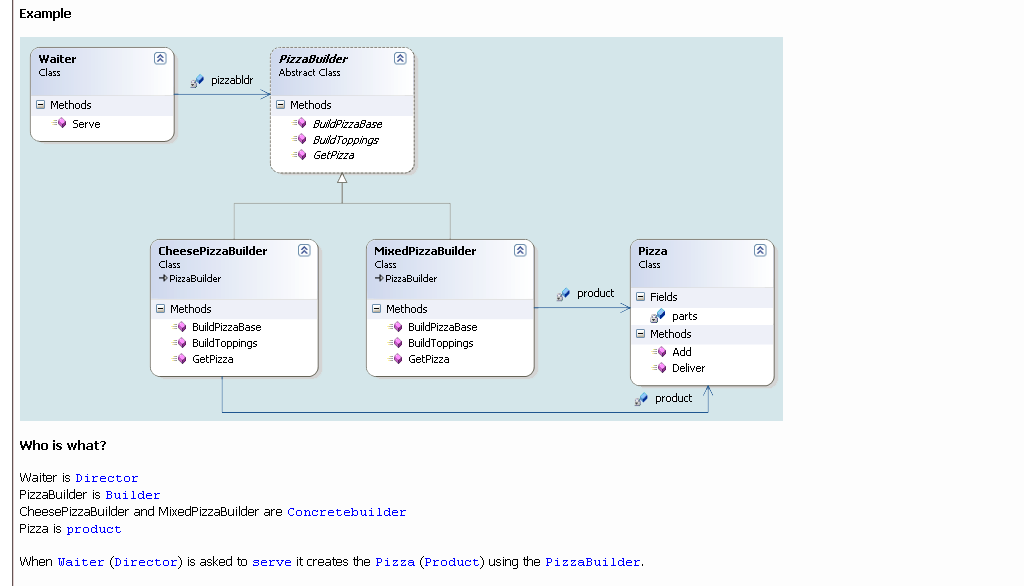
Example:

We'll define a variant of the CreateMaze member function that takes a builder of class MazeBuilder as an argument. This interface can create three things: (1) the maze, (2) rooms with a particular room number, and (3) doors between numbered rooms. The GetMaze operation returns the maze to the client. Subclasses of MazeBuilder will override this operation to return the maze that they build.

Related Patterns:

Abstract Factory is similar to Builder in that it too may construct complex objects. The primary difference is that the Builder pattern focuses on constructing a complex object step by step. Abstract Factory's emphasis is on families of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory pattern is concerned, the product gets returned immediately.  
A Composite is what the builder often builds.





BuildPizzaBase(), BuildToppings() and GetPizza() can be private virtual functions.

**Factory method:**

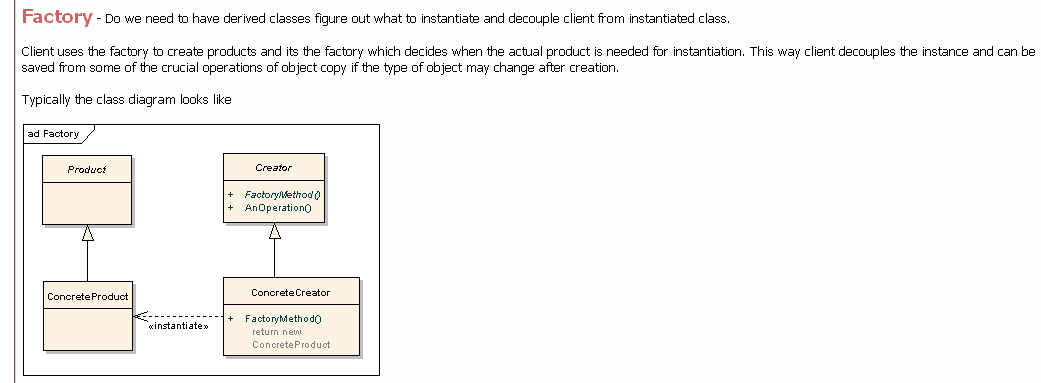
|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Factory Method | Virtual Constructor | Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. |

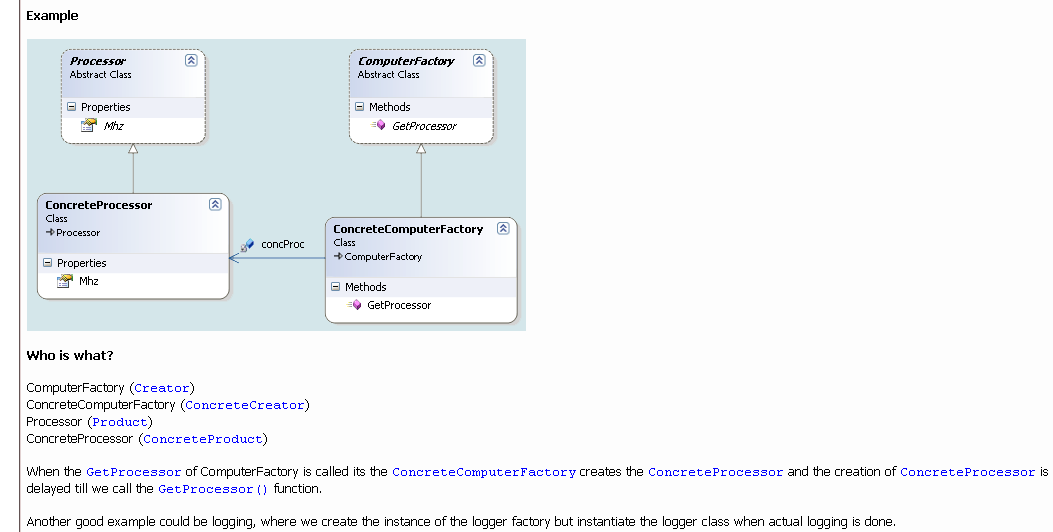
Example:

Consider a framework for applications that can present multiple documents to the user. Two key abstractions in this framework are the classes Application and Document. Both classes are abstract, and clients have to subclass them to realize their application-specific implementations. To create a drawing application, for example, we define the classes DrawingApplication and DrawingDocument. The Application class is responsible for managing Documents and will create them as required—when the user selects Open or New from a menu, for example.

Related Patterns:

Abstract Factory is often implemented with factory methods. The Motivation example in the Abstract Factory pattern illustrates Factory Method as well.  
Factory methods are usually called within Template Methods (325). In the document example above, NewDocument is a template method.  
Prototypes (117) don't require subclassing Creator. However, they often require an Initialize operation on the Product class. Creator uses Initialize to initialize the object. Factory Method doesn't require such an operation.





class Stooge {

public:

// Factory Method

static Stooge\* make\_stooge( int choice );

virtual void slap\_stick() = 0;

};

int main( void ) {

vector roles;

int choice;

while (true) {

cout << "Larry(1) Moe(2) Curly(3) Go(0): ";

cin >> choice;

if (choice == 0)

break;

roles.push\_back( Stooge::make\_stooge( choice ) );

}

for (int i=0; i < roles.size(); i++)

roles[i]->slap\_stick();

for (int i=0; i < roles.size(); i++)

delete roles[i];

}

class Larry : public Stooge {

public:

void slap\_stick() {

cout << "Larry: poke eyes\n"; }

};

class Moe : public Stooge {

public:

void slap\_stick() {

cout << "Moe: slap head\n"; }

};

class Curly : public Stooge {

public:

void slap\_stick() {

cout << "Curly: suffer abuse\n"; }

};

Stooge\* Stooge::make\_stooge( int choice ) {

if (choice == 1)

return new Larry;

else if (choice == 2)

return new Moe;

else

return new Curly;

}

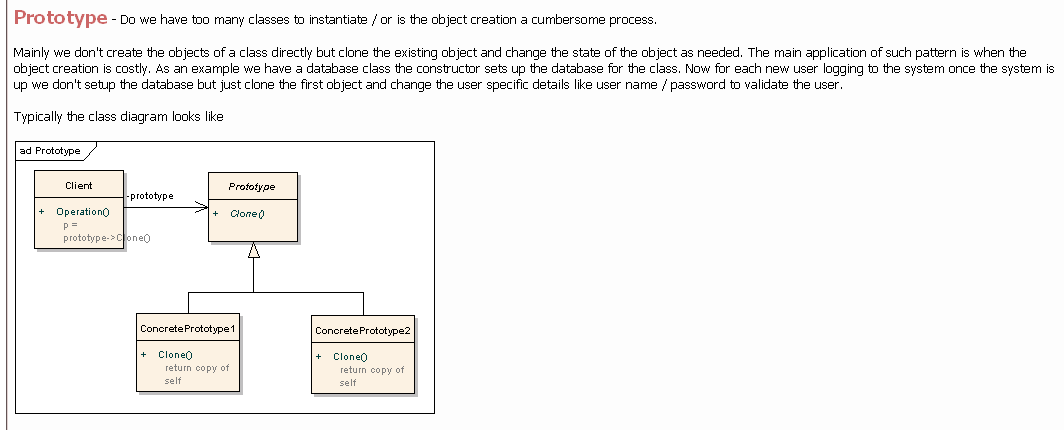
**Prototype:**

|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Prototype | prototype->clone() | Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. |

Example:

Related Patterns:

Prototype and Abstract Factory are competing patterns in some ways, as we discuss at the end of this chapter. They can also be used together, however. An Abstract Factory might store a set of prototypes from which to clone and return product objects.  
Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well.



class Stooge {

public:

virtual Stooge\* clone() = 0;

virtual void slap\_stick() = 0;

};

class Factory {

public:

static Stooge\* make\_stooge( int choice );

private:

static Stooge\* s\_prototypes[4];

};

int main( void ) {

vector roles;

int choice;

while (true) {

cout << "Larry(1) Moe(2) Curly(3) Go(0): ";

cin >> choice;

if (choice == 0)

break;

roles.push\_back(

Factory::make\_stooge( choice ) );

}

for (int i=0; i < roles.size(); ++i)

roles[i]->slap\_stick();

for (int i=0; i < roles.size(); ++i)

delete roles[i];

}

class Larry : public Stooge {

public:

Stooge\* clone() { return new Larry; }

void slap\_stick() {

cout << "Larry: poke eyes\n"; }

};

class Moe : public Stooge {

public:

Stooge\* clone() { return new Moe; }

void slap\_stick() {

cout << "Moe: slap head\n"; }

};

class Curly : public Stooge {

public:

Stooge\* clone() { return new Curly; }

void slap\_stick() {

cout << "Curly: suffer abuse\n"; }

};

Stooge\* Factory::s\_prototypes[] = {

0, new Larry, new Moe, new Curly

};

Stooge\* Factory::make\_stooge( int choice ) {

return s\_prototypes[choice]->clone();

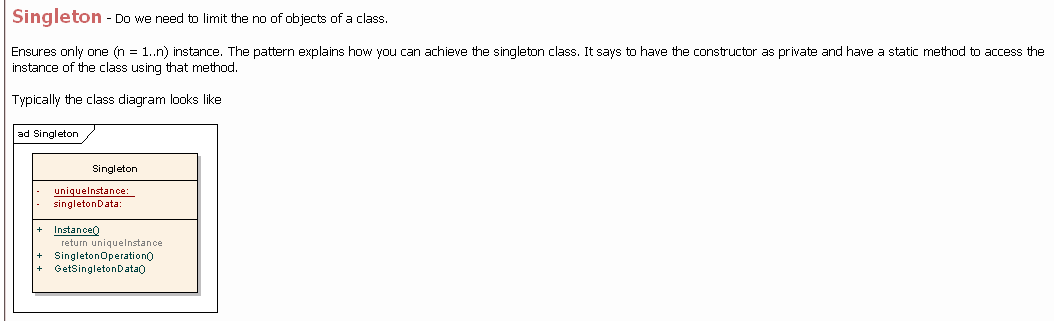
}

**Singleton:**

|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Singleton |  | Ensure a class only has one instance, and provide a global point of access to it. |

Related Patterns:

Many patterns can be implemented using the Singleton pattern. See Abstract Factory , Builder , and Prototype.

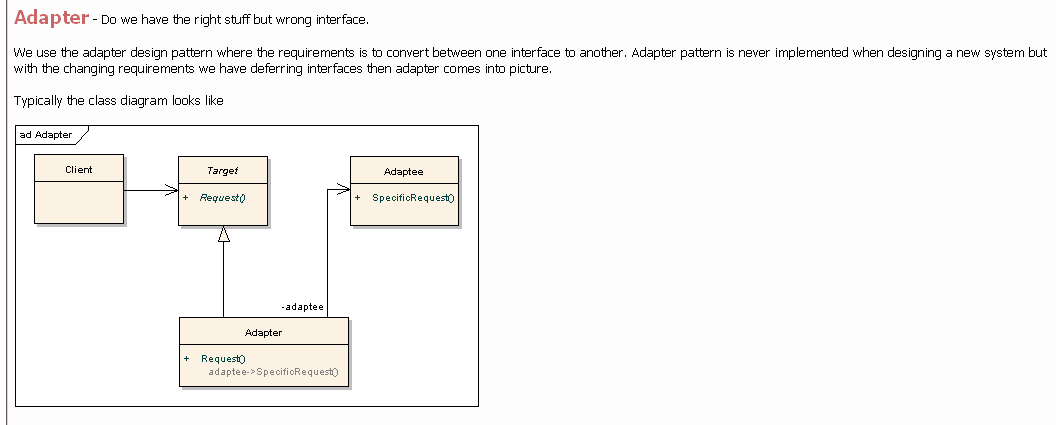




**Structural:**

**Adapter:**

|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Adapter | Wrapper | Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. |



#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:**

|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Bridge | abstraction -> implementation…..Handle/Body | Decouple an abstraction from its implementation so that the two can vary independently. |

Example:

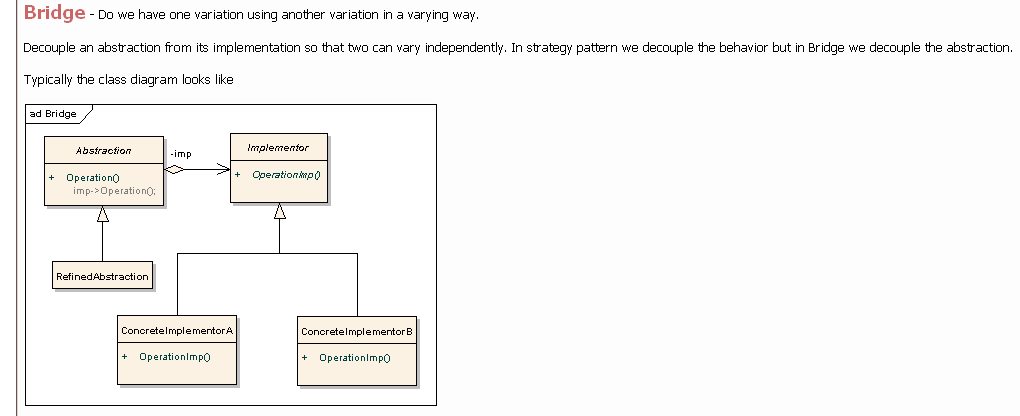
Bridge to allow multiple windowing platforms. In case study, …Supporting multiple window systems. Different look-and-feel standards are usually implemented on different window systems. Lexi's design should be as independent of the window system as possible.   
Window - ApplicationWindow, IconWindow, DialogWindow  
WindowImpl - MotifWindowImpl, MacWindowImpl, OMWindowImpl.

~~Vehicles: car, bus, 3wheeler~~

~~Engines: petrol, diesel, CNG~~

Related Patterns:

An Abstract Factory can create and configure a particular Bridge.  
The Adapter pattern is geared toward making unrelated classes work together. It is usually applied to systems after they're designed. Bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.



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

**Composite:**

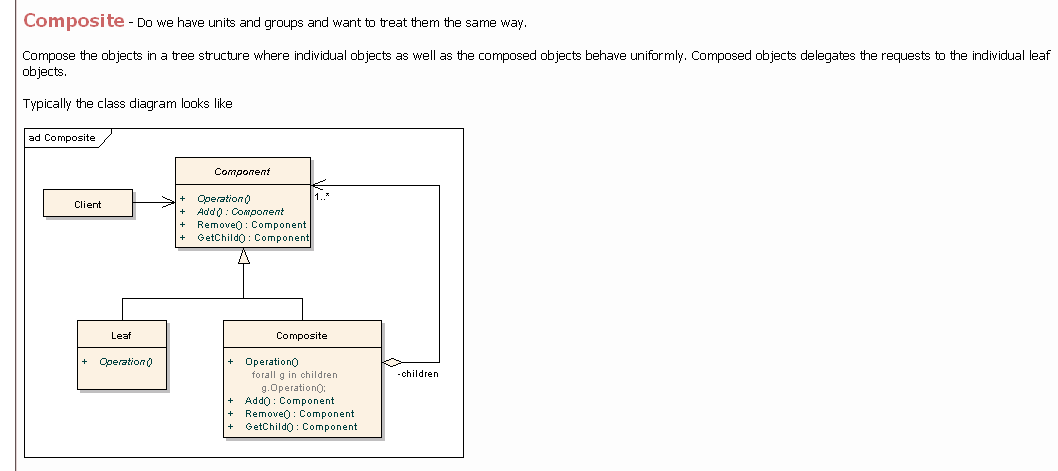
|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Composite | reverse composition | Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. |

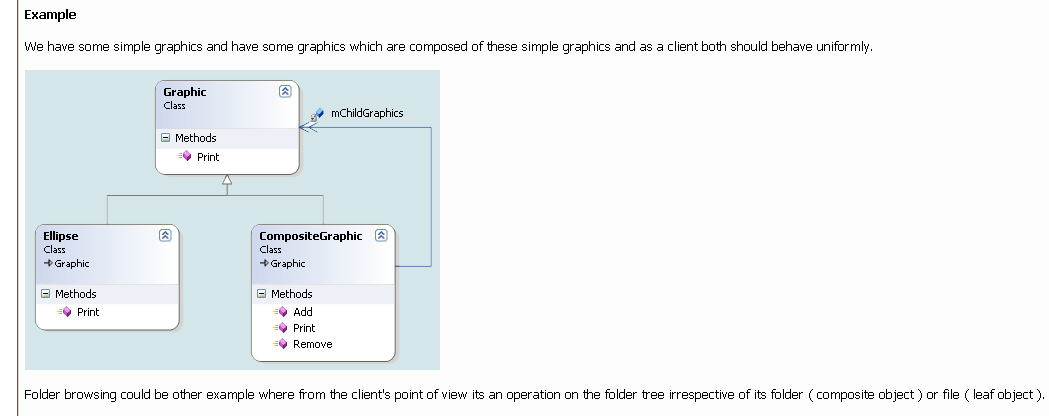
Example:

Composite to represent the document's physical structure. In case study, …Document structure. The choice of internal representation for the document affects nearly every aspect of Lexi's design. All editing, formatting, displaying, and textual analysis will require traversing the representation. The way we organize this information will impact the design of the rest of the application.   
A common way to represent hierarchically structured information is through a technique called recursive composition, which entails building increasingly complex elements out of simpler ones. Recursive composition gives us a way to compose a document out of simple graphical elements.

Related Patterns:

Often the component-parent link is used for a Chain of Responsibility.  
Decorator is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.  
Flyweight lets you share components, but they can no longer refer to their parents.  
Iterator can be used to traverse composites.  
Visitor localizes operations and behavior that would otherwise be distributed across Composite and Leaf classes.





#include <iostream>

#include <vector>

#include <string>

using std::cout;

using std::vector;

using std::string;

class Component

{

public:

virtual void list() const = 0;

virtual ~Component(){};

};

class Leaf : public Component

{

public:

explicit Leaf(int val) : value\_(val)

{

}

void list() const

{

cout << " " << value\_ << "\n";

}

private:

int value\_;

};

class Composite : public Component

{

public:

explicit Composite(string id) : id\_(id)

{

}

void add(Component \*obj)

{

table\_.push\_back(obj);

}

void list() const

{

cout << id\_ << ":" << "\n";

for (vector<Component\*>::const\_iterator it = table\_.begin();

it != table\_.end(); ++it)

{

(\*it)->list();

}

}

private:

vector <Component\*> table\_;

string id\_;

};

int main()

{

Leaf num0(0);

Leaf num1(1);

Leaf num2(2);

Leaf num3(3);

Leaf num4(4);

Composite container1("Container 1");

Composite container2("Container 2");

container1.add(&num0);

container1.add(&num1);

container2.add(&num2);

container2.add(&num3);

container2.add(&num4);

container1.add(&container2);

container1.list();

return 0;

}

**Decorator:**

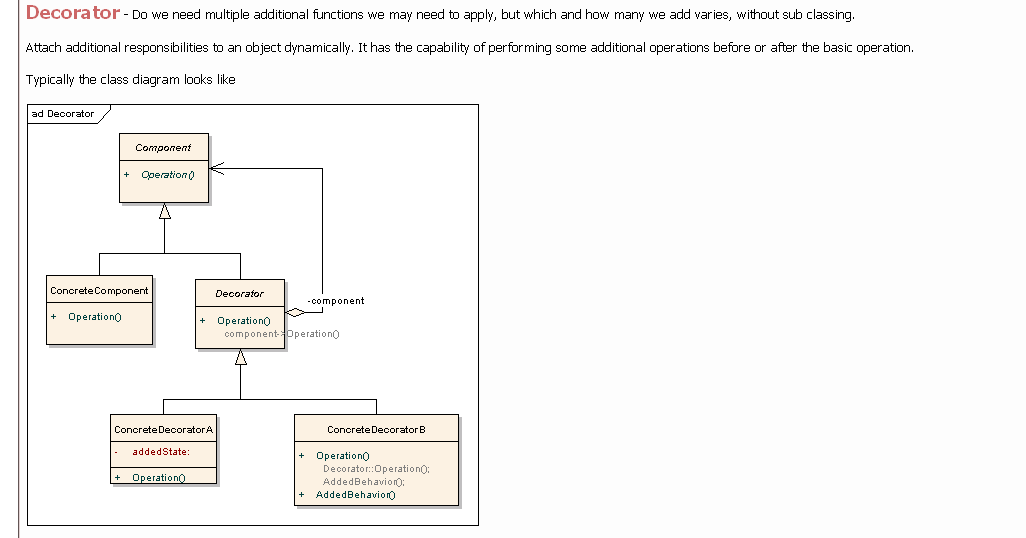
|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Decorator | Wrapper | Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. |

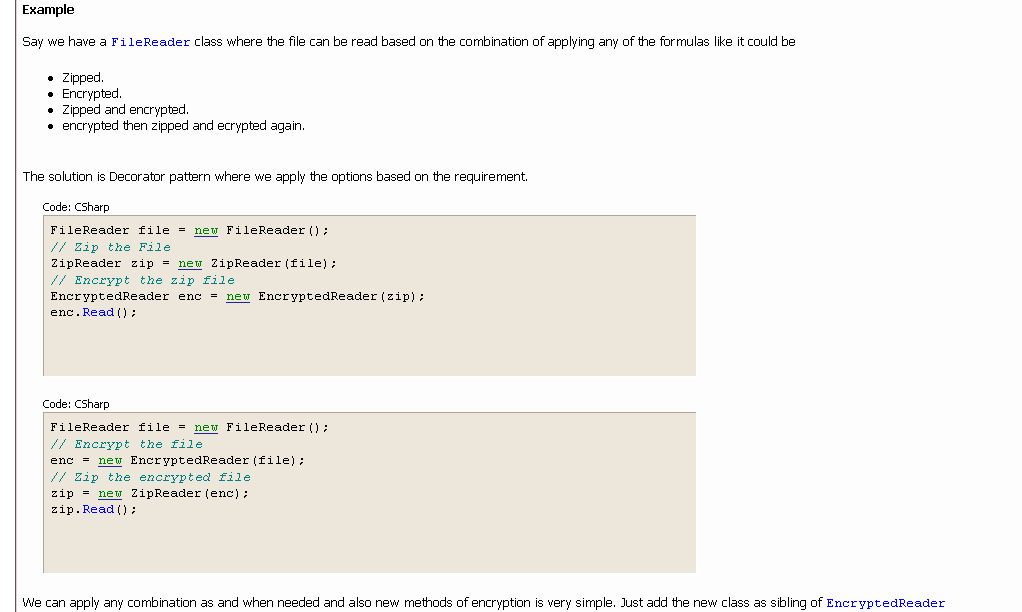
Example:

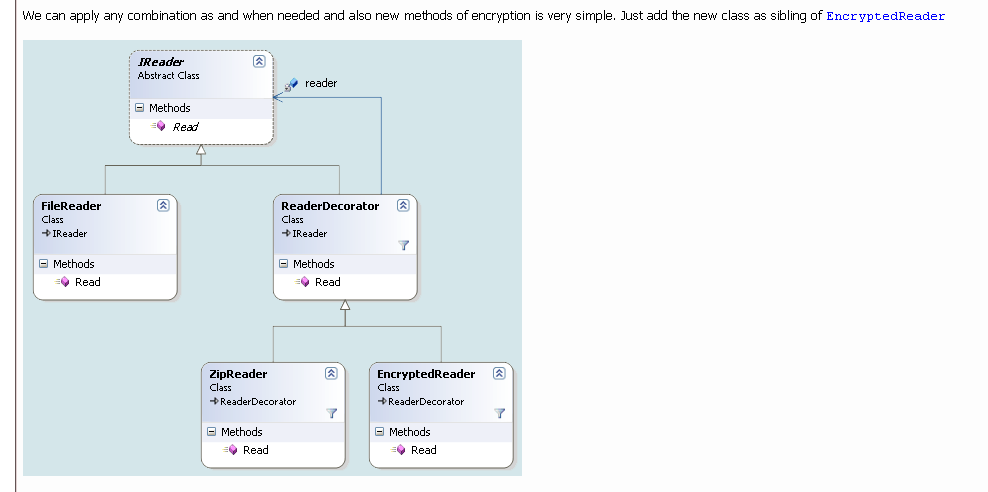
Decorator for embellishing the user interface. In case study, …Embellishing the user interface. Lexi's user interface includes scroll bars, borders, and drop shadows that embellish the WYSIWYG document interface. Such embellishments are likely to change as Lexi's user interface evolves. Hence it's important to be able to add and remove embellishments easily without affecting the rest of the application.

Related Patterns:

Adapter : A decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface.  
Composite : 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.  
Strategy : A decorator lets you change the skin of an object; a strategy lets you change the guts. These are two alternative ways of changing an object.







class I {

public:

virtual ~I() { }

virtual void do\_it() = 0;

};

class A : public I {

public:

~A() { cout << "A dtor" << '\n'; }

/\*virtual\*/ void do\_it() { cout << 'A'; }

};

class D : public I {

public:

D( I\* inner ) { m\_wrappee = inner; }

~D() { delete m\_wrappee; }

/\*virtual\*/ void do\_it() { m\_wrappee->do\_it(); }

private:

I\* m\_wrappee;

};

class X : public D {

public:

X( I\* core ) : D( core ) { }

~X() { cout << "X dtor" << " "; }

/\*virtual\*/ void do\_it() {

D::do\_it();

cout << 'X';

}

};

class Y : public D {

public:

Y( I\* core ) : D( core ) { }

~Y() { cout << "Y dtor" << " "; }

/\*virtual\*/ void do\_it() {

D::do\_it();

cout << 'Y';

}

};

class Z : public D {

public:

Z( I\* core ) : D( core ) { }

~Z() { cout << "Z dtor" << " "; }

/\*virtual\*/ void do\_it() {

D::do\_it();

cout << 'Z';

}

};

int 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->do\_it(); cout << '\n';

anXY->do\_it(); cout << '\n';

anXYZ->do\_it(); 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

**Facade:**

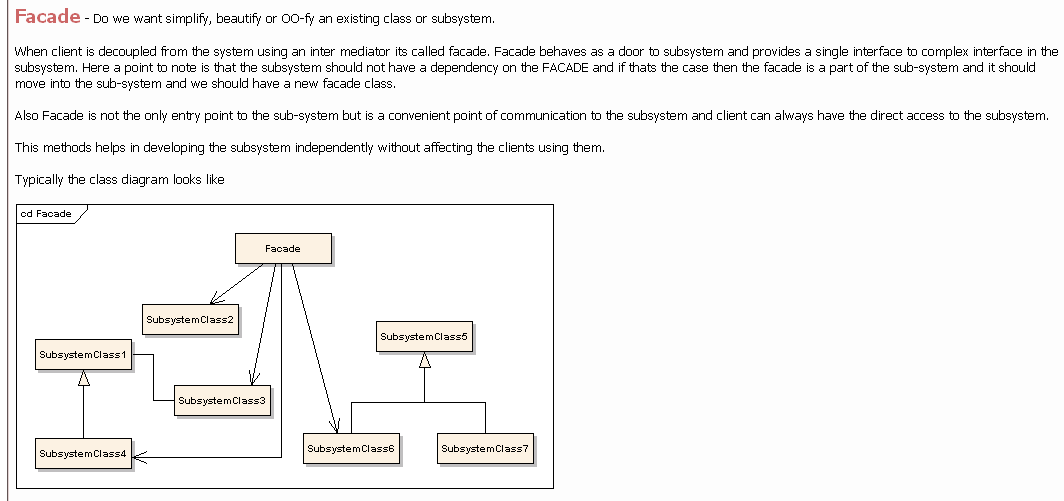
|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Façade | Common interface | Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. |

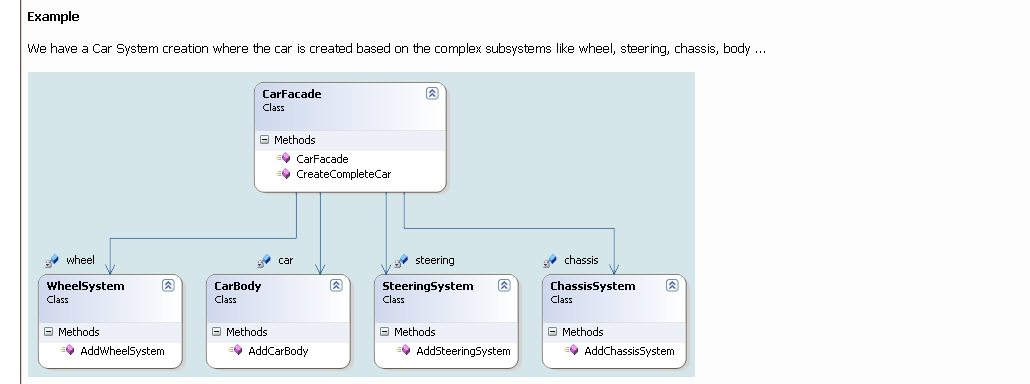
Example:

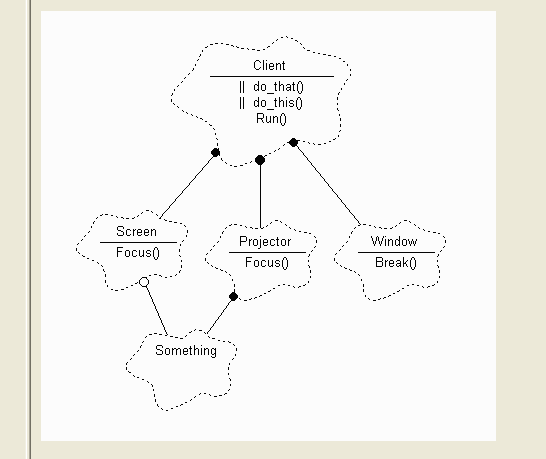
Consider for example a programming environment that gives applications access to its compiler subsystem. This subsystem contains classes such as Scanner, Parser, ProgramNode, BytecodeStream, and ProgramNodeBuilder that implement the compiler. Some specialized applications might need to access these classes directly. But most clients of a compiler generally don't care about details like parsing and code generation; they merely want to compile some code. For them, the powerful but low-level interfaces in the compiler subsystem only complicate their task.

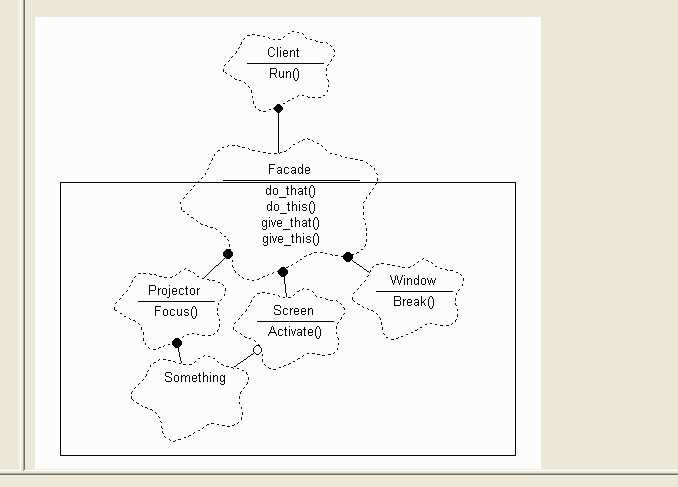
Related Patterns:

Abstract Factory can be used with Facade to provide an interface for creating subsystem objects in a subsystem-independent way. Abstract Factory can also be used as an alternative to Facade to hide platform-specific classes.  
Mediator is similar to Facade in that it abstracts functionality of existing classes. However, Mediator's purpose is to abstract arbitrary communication between colleague objects, often centralizing functionality that doesn't belong in any one of them. A mediator's colleagues are aware of and communicate with the mediator instead of communicating with each other directly. In contrast, a facade merely abstracts the interface to subsystem objects to make them easier to use; it doesn't define new functionality, and subsystem classes don't know about it.  
Usually only one Facade object is required. Thus Facade objects are often Singletons.









1. The compilers that we use everyday to process the computer code that we

have written is a prime example of the facade pattern in action.

#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

**Flyweight:**

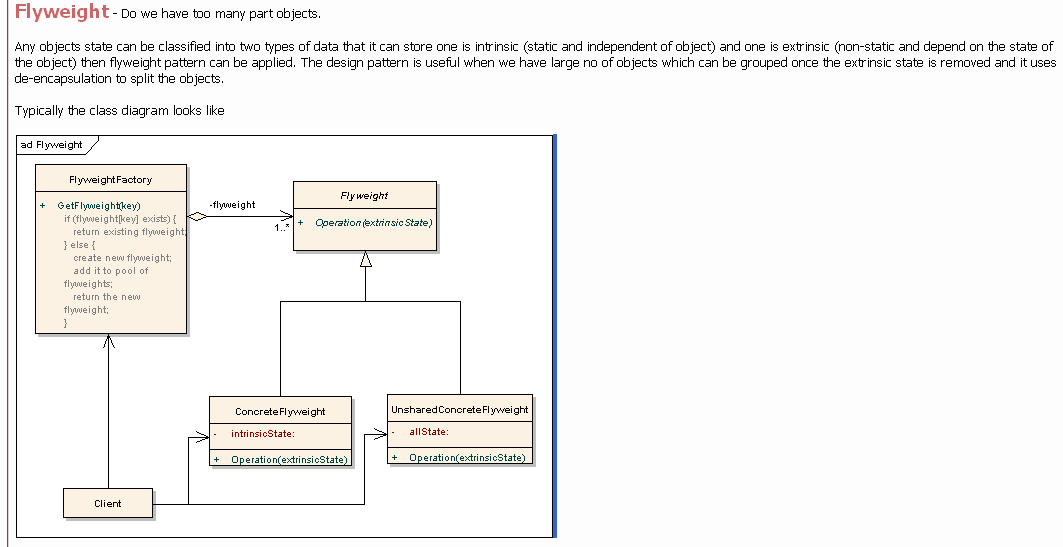
|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Flyweight | shared objects | Use sharing to support large numbers of fine-grained objects efficiently. |

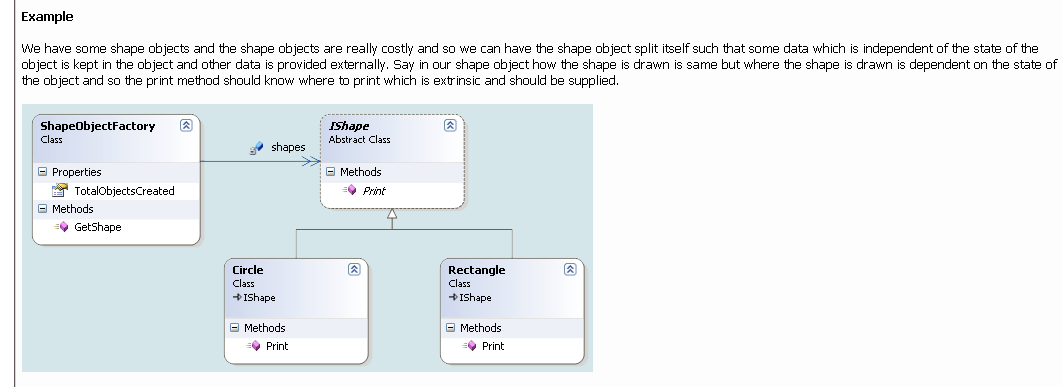
Example:

For example, most document editor implementations have text formatting and editing facilities that are modularized to some extent. Object-oriented document editors typically use objects to represent embedded elements like tables and figures. However, they usually stop short of using an object for each character in the document, even though doing so would promote flexibility at the finest levels in the application. Characters and embedded elements could then be treated uniformly with respect to how they are drawn and formatted. The application could be extended to support new character sets without disturbing other functionality. The application's object structure could mimic the document's physical structure. The following diagram shows how a document editor can use objects to represent characters.  
The drawback of such a design is its cost. Even moderate-sized documents may require hundreds of thousands of character objects, which will consume lots of memory and may incur unacceptable run-time overhead. The Flyweight pattern describes how to share objects to allow their use at fine granularities without prohibitive cost.  
A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context—it's indistinguishable from an instance of the object that's not shared. Flyweights cannot make assumptions about the context in which they operate. The key concept here is the distinction between intrinsic and extrinsic state. Intrinsic state is stored in the flyweight; it consists of information that's independent of the flyweight's context, thereby making it sharable. Extrinsic state depends on and varies with the flyweight's context and therefore can't be shared. Client objects are responsible for passing extrinsic state to the flyweight when it needs it.  
Logically there is an object for every occurrence of a given character in the document. Physically, however, there is one shared flyweight object per character, and it appears in different contexts in the document structure. Each occurrence of a particular character object refers to the same instance in the shared pool of flyweight objects:

Related Patterns:

Related Patterns  
The Flyweight pattern is often combined with the Composite pattern to implement a logically hierarchical structure in terms of a directed-acyclic graph with shared leaf nodes.  
  
It's often best to implement State and Strategy objects as flyweights.





// Before

// 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 client supply it when methods are called.

// This places more responsibility on the client,

// but, considerably fewer instances of the Fly-

// weight class are now created. Sharing of these

// instances is facilitated by introducing a Fac-

// tory 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 sup-

// plied by the client when report() is called).

class Gazillion {

public:

Gazillion() {

m\_value\_one = s\_num / Y;

m\_value\_two = s\_num % Y;

++s\_num;

}

void report() {

cout << m\_value\_one << m\_value\_two << ' ';

}

static int X, Y;

private:

int m\_value\_one;

int m\_value\_two;

static int s\_num;

};

int Gazillion::X = 6, Gazillion::Y = 10,

Gazillion::s\_num = 0;

int main( void ) {

Gazillion matrix[Gazillion::X][Gazillion::Y];

for (int i=0; i < Gazillion::X; ++i) {

for (int j=0; j < Gazillion::Y; ++j)

matrix[i][j].report();

cout << '\n';

} }

// 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

// After

class Gazillion {

public:

Gazillion( int value\_one ) {

m\_value\_one = value\_one;

cout << "ctor: "<< m\_value\_one << '\n';

}

~Gazillion() {

cout << m\_value\_one << ' ';

}

void report( int value\_two ) {

cout << m\_value\_one << value\_two << ' ';

}

private:

int m\_value\_one;

};

class Factory {

public:

static Gazillion\* get\_fly( int in ) {

if ( ! s\_pool[in])

s\_pool[in] = new Gazillion( in );

return s\_pool[in];

}

static void clean\_up() {

cout << "dtors: ";

for (int i=0; i < X; ++i)

if (s\_pool[i])

delete s\_pool[i];

cout << '\n';

}

static int X, Y;

private:

static Gazillion\* s\_pool[];

};

int Factory::X = 6, Factory::Y = 10;

Gazillion\* Factory::s\_pool[] = { 0,0,0,0,0,0 };

int main( void ) {

for (int i=0; i < Factory::X; ++i) {

for (int j=0; j < Factory::Y; ++j)

Factory::get\_fly(i)->report(j);

cout << '\n';

}

Factory::clean\_up();

}

// 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

**Proxy:**

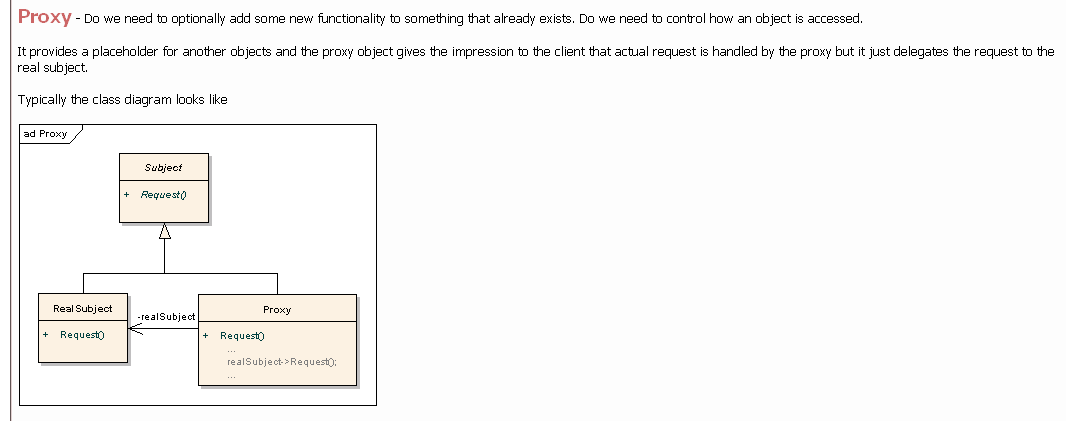
|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Proxy | Defer the creation and display of large n complex images in document …smart pointers….Surrogate | Provide a surrogate or placeholder for another object to control access to it |

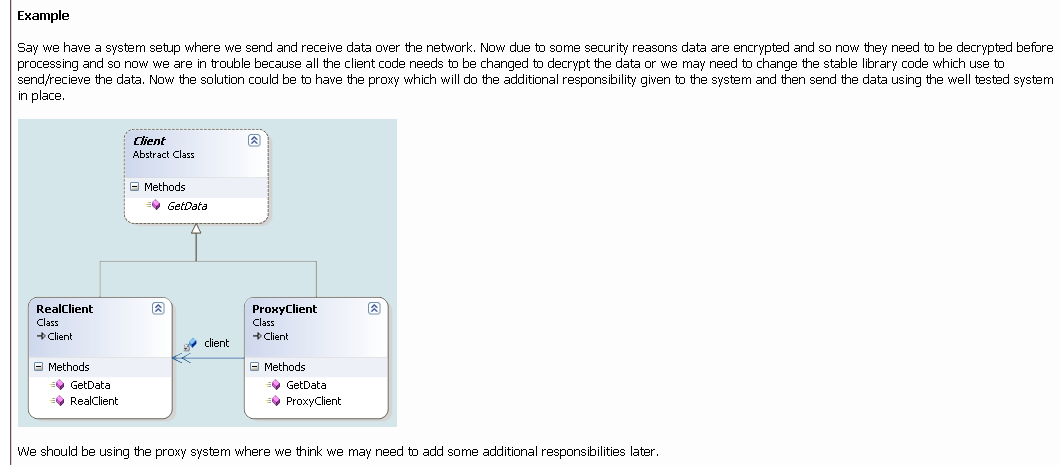
Example:

In case study...Defer the creation and display of large n complex images in document editor. Do it on demand. Principals: copy-on-write, lazy initialization etc. Other examples: smart pointers

Related Patterns:

Adapter : An adapter provides a different interface to the object it adapts. In contrast, a proxy provides the same interface as its subject. However, a proxy used for access protection might refuse to perform an operation that the subject will perform, so its interface may be effectively a subset of the subject's.  
  
Decorator : Although decorators can have similar implementations as proxies, decorators have a different purpose. A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.  
  
Proxies vary in the degree to which they are implemented like a decorator. A protection proxy might be implemented exactly like a decorator. On the other hand, a remote proxy will not contain a direct reference to its real subject but only an indirect reference, such as "host ID and local address on host." A virtual proxy will start off with an indirect reference such as a file name but will eventually obtain and use a direct reference.





// Initialization on first use

// 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 m\_id;

public:

RealImage( int i ) { m\_id = i;

cout << " $$ ctor: "<< m\_id << '\n'; }

~RealImage() {

cout << " dtor: " << m\_id << '\n'; }

void draw() {

cout << " drawing image " << m\_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\* m\_the\_real\_thing;

int m\_id;

static int s\_next;

public:

Image() {

m\_id = s\_next++;

// 3. Initialized to null

m\_the\_real\_thing = 0;

}

~Image() { delete m\_the\_real\_thing; }

void draw() {

// 4. When a request comes in, the real object is

// created "on first use"

if ( ! m\_the\_real\_thing)

m\_the\_real\_thing = new RealImage( m\_id );

// 5. The request is always delegated

m\_the\_real\_thing->draw();

} };

int Image::s\_next = 1;

int main( void ) {

Image images[5];

for (int i; true; ) {

cout << "Exit[0], Image[1-5]: ";

cin >> i;

if (i == 0)

break;

images[i-1].draw();

} }

// 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]: 0

// dtor: 4

// dtor: 2

**Behavioural:**

**Chain of responsibility:**

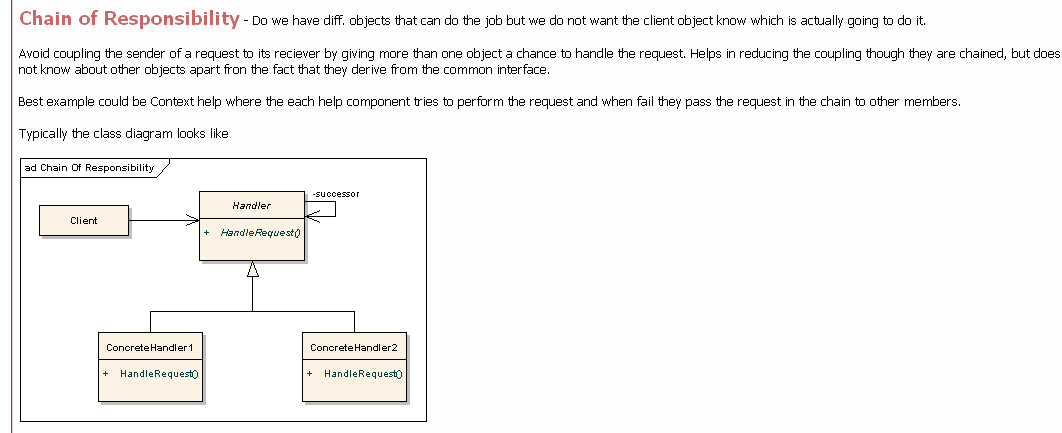
|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Chain of Responsibility | Help Handler | Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it. |

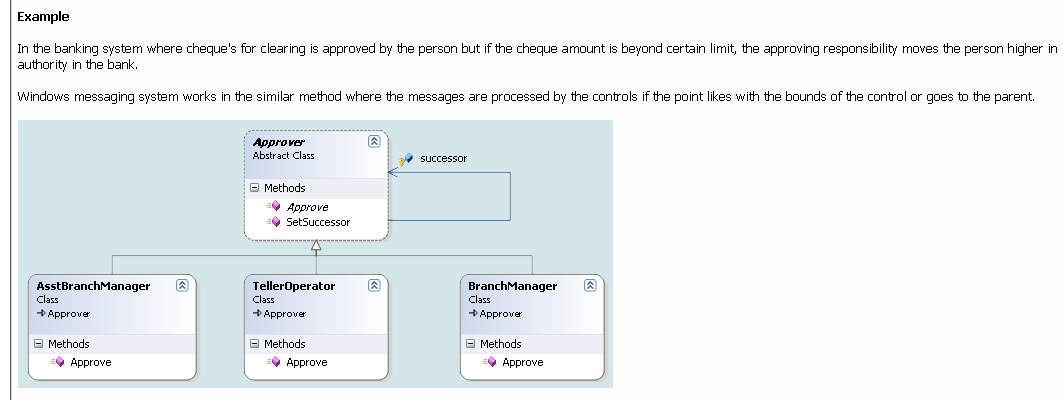
Example:

When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for handling it. E.g. Help Handler

Related Patterns:

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





// The approach of the five-year-old.

#include <iostream>

#include <vector>

#include "../purge.h"

using namespace std;

enum Answer { NO, YES };

class GimmeStrategy {

public:

virtual Answer canIHave() = 0;

virtual ~GimmeStrategy() {}

};

class AskMom : public GimmeStrategy {

public:

Answer canIHave() {

cout << "Mooom? Can I have this?" << endl;

return NO;

}

};

class AskDad : public GimmeStrategy {

public:

Answer canIHave() {

cout << "Dad, I really need this!" << endl;

return NO;

}

};

class AskGrandpa : public GimmeStrategy {

public:

Answer canIHave() {

cout << "Grandpa, is it my birthday yet?" << endl;

return NO;

}

};

class AskGrandma : public GimmeStrategy {

public:

Answer canIHave() {

cout << "Grandma, I really love you!" << endl;

return YES;

}

};

class Gimme : public GimmeStrategy {

vector<GimmeStrategy\*> chain;

public:

Gimme() {

chain.push\_back(new AskMom());

chain.push\_back(new AskDad());

chain.push\_back(new AskGrandpa());

chain.push\_back(new AskGrandma());

}

Answer canIHave() {

vector<GimmeStrategy\*>::iterator it = chain.begin();

while(it != chain.end())

if((\*it++)->canIHave() == YES)

return YES;

// Reached end without success...

cout << "Whiiiiinnne!" << endl;

return NO;

}

~Gimme() { purge(chain); }

};

int main() {

Gimme chain;

chain.canIHave();

} ///:~

// Notice that the Context class Gimme and all the Strategy classes are all derived from the same base class, GimmeStrategy.

// If you study the section on Chain of Responsibility in GoF, you ll find that the structure differs significantly from the

// one above because they focus on creating their own linked list. However, if you keep in mind that the essence of Chain of

// Responsibility is to try a number of solutions until you find one that works, you ll realize that the implementation of

// the sequencing mechanism is not an essential part of the pattern.

Example 2:

// 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

**Command:**

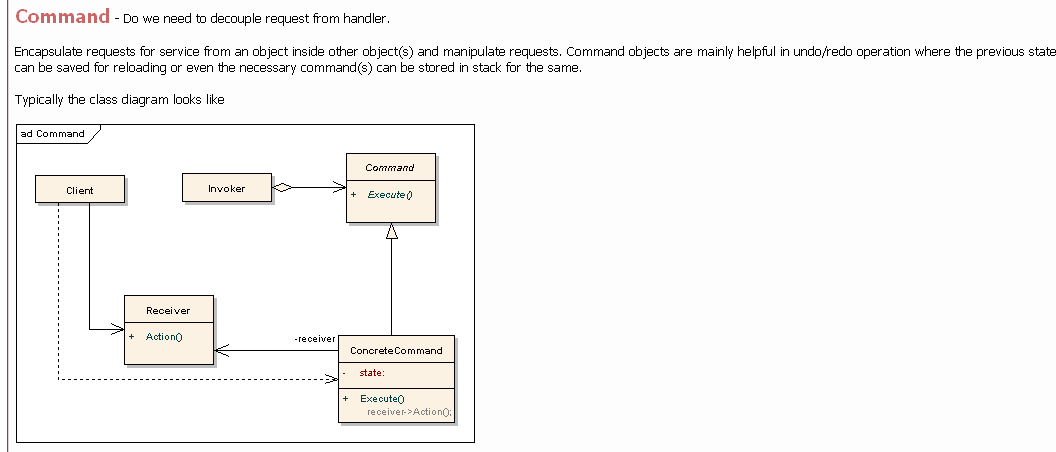
|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Command | encapsulate user actions….Action, Transaction | Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. |

Example:

Command for undoable user operations. In case study, ... User operations. Users control Lexi through various user interfaces, including buttons and pull-down menus. The functionality behind these interfaces is scattered throughout the objects in the application. The challenge here is to provide a uniform mechanism both for accessing this scattered functionality and for undoing its effects.   
parameterize objects by an action to perform. You can express such parameterization in a procedural language with a callback function, that is, a function that's registered somewhere to be called at a later point. Commands are an object-oriented replacement for callbacks.

Related Patterns:

A Composite can be used to implement MacroCommands.  
A Memento can keep state the command requires to undo its effect.  
A command that must be copied before being placed on the history list acts as a Prototype.



In object-oriented programming, the Command pattern is a design pattern in

which objects are used to represent actions. A command object encapsulates an

action and its parameters.

For example, a printing library might include a PrintJob class. A user would

typically create a new PrintJob object, set its properties (the document to be

printed, the number of copies, and so on), and finally call a method to send

the job to the printer.

In this case, the same functionality could be exposed via a single

SendJobToPrinter() procedure with many parameters. As it takes more code to

write a command class than to write a procedure, there is often a reason to use

a class. There are many possible reasons:

- A command object is a convenient temporary storage for procedure parameters.

It can be used while assembling the parameters for a function call and allows

the command to be set aside for later use.

- A class is a convenient place to collect code and data related to a command.

A command object can hold information about the command, such as its name or

which user launched it; and answer questions about it, such as how long it

will likely take. It also allows the command to be executed some time after

it is defined.

- Treating commands as objects enables data structures containing multiple

commands. A complex process could be treated as a tree or graph of command

objects.

- Treating commands as objects supports undo-able operations, provided that the

command objects are stored (for example in a stack).

Command objects are useful for implementing:

1. Multi-level undo

If all user actions in a program are implemented as command objects, the

program can keep a stack of the most recently executed commands. When the user

wants to undo a command, the program simply pops the most recent command

object and executes its undo() method.

2. Transactional behavior

Undo is perhaps even more essential when it's called rollback and happens

automatically when an operation fails partway through. Installers need this

and so do databases. Command objects can also be used to implement two-phase

commit.

3. Progress bars

Suppose a program has a sequence of commands that it executes in order.

If each command object has a getEstimatedDuration() method, the program can

easily estimate the total duration. It can show a progress bar that

meaningfully reflects how close the program is to completing all the tasks.

4. Wizards

Often a wizard presents several pages of configuration for a single action

that happens only when the user clicks the "Finish" button on the last page.

In these cases, a natural way to separate user interface code from application

code is to implement the wizard using a command object. The command object is

created when the wizard is first displayed. Each wizard page stores its GUI

changes in the command object, so the object is populated as the user

progresses. "Finish" simply triggers a call to execute(). This way, the

command class contains no user interface code.

Example:

// Consider a simple recipe making program. Every command could be represented as an object.

// Then if the user makes a mistake then he/she can undo the command by removing that object

// from the stack.

#include <iostream>

#include <vector>

#include <string>

using namespace std;

class Command{

public:

virtual void execute(void) =0;

virtual ~Command(void){};

};

class Ingredient : public Command {

public:

Ingredient(string amount, string ingredient){

\_ingredient = ingredient;

\_amount = amount;

}

void execute(void){

cout << " \*Add " << \_amount << " of " << \_ingredient << endl;

}

private:

string \_ingredient;

string \_amount;

};

class Step : public Command {

public:

Step(string action, string time){

\_action= action;

\_time= time;

}

void execute(void){

cout << " \*" << \_action << " for " << \_time << endl;

}

private:

string \_time;

string \_action;

};

class CmdStack{

public:

void add(Command \*c) {

commands.push\_back(c);

}

void createRecipe(void){

for(vector<Command\*>::size\_type x=0;x<commands.size();x++){

commands[x]->execute();

}

}

void undo(void){

if(commands.size() > 0) {

commands.pop\_back();

}

else {

cout << "Can't undo" << endl;

}

}

private:

vector<Command\*> commands;

};

int main(void) {

CmdStack list;

//Create ingredients

Ingredient first("2 tablespoons", "vegetable oil");

Ingredient second("3 cups", "rice");

Ingredient third("1 bottle","Ketchup");

Ingredient fourth("4 ounces", "peas");

Ingredient fifth("1 teaspoon", "soy sauce");

//Create Step

Step step("Stir-fry","3-4 minutes");

//Create Recipe

cout << "Recipe for simple Fried Rice" << endl;

list.add(&first);

list.add(&second);

list.add(&step);

list.add(&third);

list.undo();

list.add(&fourth);

list.add(&fifth);

list.createRecipe();

cout << "Enjoy!" << endl;

return 0;

}

**Interpreter**

|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Interpreter | search strings using patterns etc | Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language. |

Example:

searching for strings that match a pattern is a common problem. Regular expressions are a standard language for specifying patterns of strings. Rather than building custom algorithms to match each pattern against strings, search algorithms could interpret a regular expression that specifies a set of strings to match.  
The Interpreter pattern describes how to define a grammar for simple languages, represent sentences in the language, and interpret these sentences.

Related Patterns:

Composite : The abstract syntax tree is an instance of the Composite pattern.  
Flyweight shows how to share terminal symbols within the abstract syntax tree.  
Iterator : The interpreter can use an Iterator to traverse the structure.  
Visitor can be used to maintain the behavior in each node in the abstract syntax tree in one class.

**Iterator:**

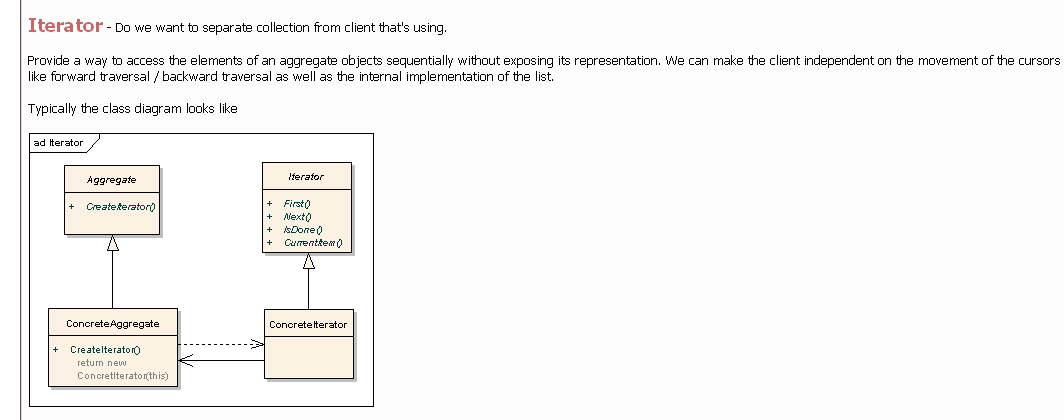
|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Iterator | traversing thru collection...Cursor | Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. |

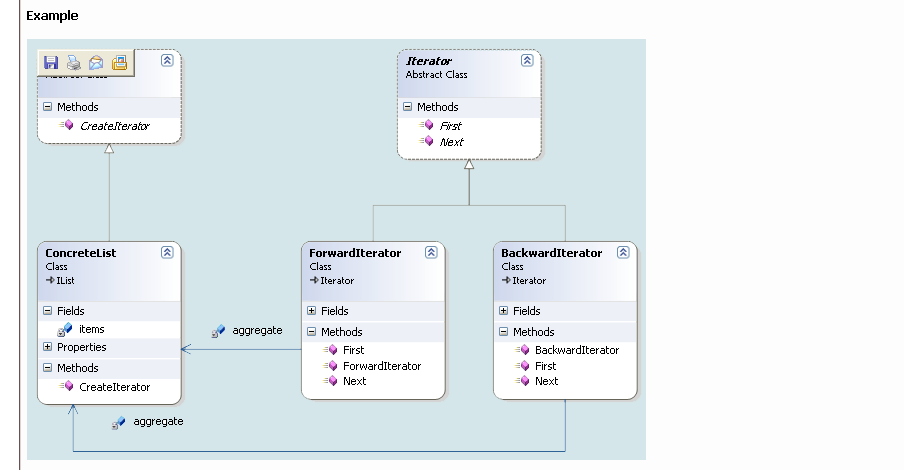
**Example:**

Iterator for accessing and traversing object structures. In case study, ...(NA)  
The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an iterator object. The Iterator class defines an interface for accessing the list's elements. An iterator object is responsible for keeping track of the current element; that is, it knows which elements have been traversed already.

**Related Patterns:**

Composite : Iterators are often applied to recursive structures such as Composites.  
Factory Method : Polymorphic iterators rely on factory methods to instantiate the appropriate Iterator subclass.  
Memento is often used in conjunction with the Iterator pattern. An iterator can use a memento to capture the state of an iteration. The iterator stores the memento internally.





**Mediator:**

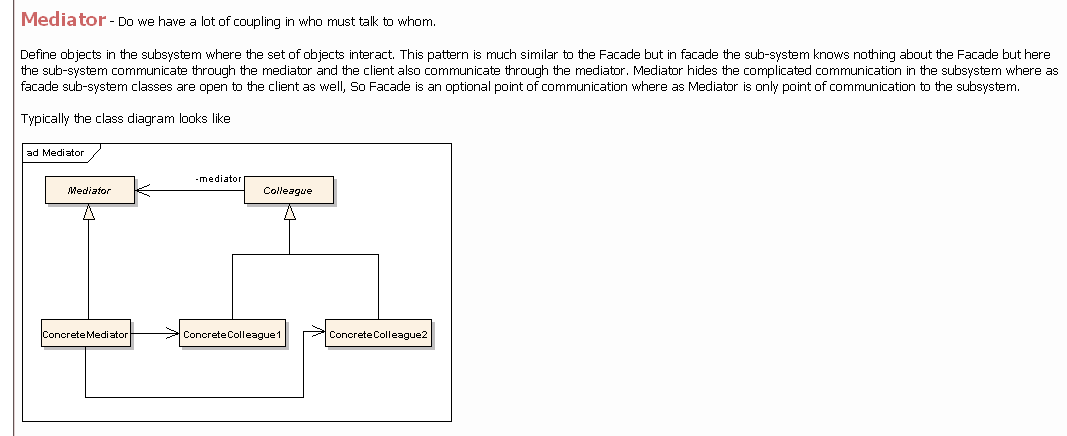
|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Mediator | help communication among dialog widgets | Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. |

**Example:**

consider the implementation of dialog boxes in a graphical user interface. A dialog box uses a window to present a collection of widgets such as buttons, menus, and entry fields. Often there are dependencies between the widgets in the dialog. For example, a button gets disabled when a certain entry field is empty. Selecting an entry in a list of choices called a list box might change the contents of an entry field. Conversely, typing text into the entry field might automatically select one or more corresponding entries in the list box. A mediator is responsible for controlling and coordinating the interactions of a group of objects. The mediator serves as an intermediary that keeps objects in the group from referring to each other explicitly. The objects only know the mediator, thereby reducing the number of interconnections.

**Related Patterns:**

Use the Mediator pattern when a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.  
  
Facade differs from Mediator in that it abstracts a subsystem of objects to provide a more convenient interface. Its protocol is unidirectional; that is, Facade objects make requests of the subsystem classes but not vice versa. In contrast, Mediator enables cooperative behavior that colleague objects don't or can't provide, and the protocol is multidirectional.  
Colleagues can communicate with the mediator using the Observer pattern.



In order to have a good object oriented design we have to create lots of

classes interacting one with each other. If certain principles are not

applied the final framework will end in a total mess where each object

relies on many other objects in order to run. In order to avoid tight

coupled frameworks, we need a mechanism to facilitate the interaction

between objects in a manner in that objects are not aware of the existence

of other objects.

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.

The mediator example is one pattern that is already used in many applications.

One of the examples is represented by the Dialog classes in GUI applications frameworks.

A Dialog window is a collection of graphic and non-graphic controls. The Dialog

class provides the mechanism to facilitate the interaction between controls.

For example, when a new value is selected from a ComboBox object a Label has

to display a new value. Both the ComboBox and the Label are not aware of each others

structure and all the interaction is managed by the Dialog object.

Each control is not aware of the existence of other controls.

Example:

// Before

// Node objects interact directly with each other,

// recursion is required, removing a Node is hard,

// and it is not possible to remove the first node.

class Node {

public:

Node( int v ) {

m\_val = v;

m\_next = 0;

}

void add\_node( Node\* n ) {

if (m\_next)

m\_next->add\_node( n );

else

m\_next = n;

}

void traverse() {

cout << m\_val << " ";

if (m\_next)

m\_next->traverse();

else

cout << '\n';

}

void remove\_node( int v ) {

if (m\_next)

if (m\_next->m\_val == v)

m\_next = m\_next->m\_next;

else

m\_next->remove\_node( v );

}

private:

int m\_val;

Node\* m\_next;

};

int main( void ) {

Node lst( 11 );

Node two( 22 ), thr( 33 ), fou( 44 );

lst.add\_node( &two );

lst.add\_node( &thr );

lst.add\_node( &fou );

lst.traverse();

lst.remove\_node( 44 );

lst.traverse();

lst.remove\_node( 22 );

lst.traverse();

lst.remove\_node( 11 );

lst.traverse();

}

// 11 22 33 44

// 11 22 33

// 11 33

// 11 33

// After

// A "mediating" List class focuses and simplifies

// all the administrative responsibilities, and the

// recursion (which does not scale up well) has been

// eliminated.

class Node {

public:

Node( int v ) { m\_val = v; }

int get\_val() { return m\_val; }

private:

int m\_val;

};

class List {

public:

void add\_node( Node\* n ) {

m\_arr.push\_back( n );

}

void traverse() {

for (int i=0; i < m\_arr.size(); ++i)

cout << m\_arr[i]->get\_val() << " ";

cout << '\n';

}

void remove\_node( int v ) {

for (vector::iterator it = m\_arr.begin();

it != m\_arr.end(); ++it)

if ((\*it)->get\_val() == v) {

m\_arr.erase( it );

break;

} }

private:

vector<Node> m\_arr;

};

int main( void ) {

List lst;

Node one( 11 ), two( 22 );

Node thr( 33 ), fou( 44 );

lst.add\_node( &one );

lst.add\_node( &two );

lst.add\_node( &thr );

lst.add\_node( &fou );

lst.traverse();

lst.remove\_node( 44 );

lst.traverse();

lst.remove\_node( 22 );

lst.traverse();

lst.remove\_node( 11 );

lst.traverse();

}

// 11 22 33 44

// 11 22 33

// 11 33

// 33

**Memento:**

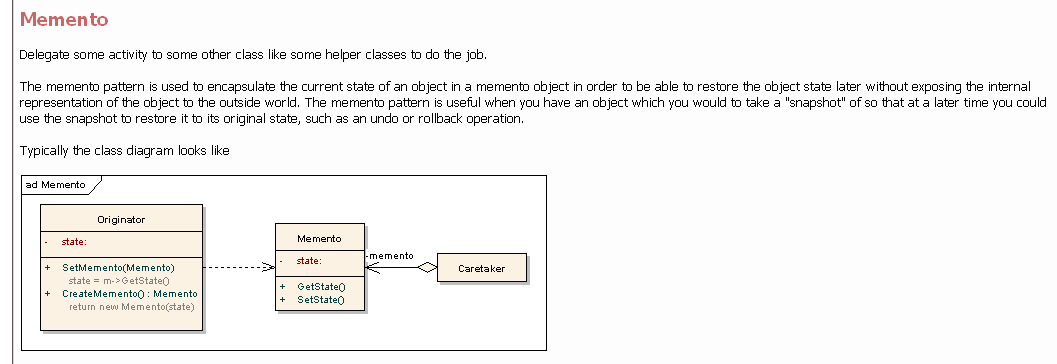
|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Memento | Remember state to be restored by undo….Token | Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later. |

Example:

Consider for example a graphical editor that supports connectivity between objects. A user can connect two rectangles with a line, and the rectangles stay connected when the user moves either of them. The editor ensures that the line stretches to maintain the connection. ConstraintSolver uses the results of its calculations to rearrange the graphics so that they maintain the proper connections.   
A memento is an object that stores a snapshot of the internal state of another object—the memento's originator. The undo mechanism will request a memento from the originator when it needs to checkpoint the originator's state. The originator initializes the memento with information that characterizes its current state. Only the originator can store and retrieve information from the memento—the memento is "opaque" to other objects.

Related Patterns:

Command : Commands can use mementos to maintain state for undoable operations.  
Iterator : Mementos can be used for iteration as described earlier.



// "check point" and "roll back" a Stack

class Memento {

friend class Stack;

int \*m\_items, m\_num;

Memento( int\* arr, int num ) {

m\_items = new int[m\_num = num];

for (int i=0; i < m\_num; i++)

m\_items[i] = arr[i];

}

public:

~Memento() { delete m\_items; }

};

class Stack {

int m\_items[10], m\_sp;

public:

Stack() { m\_sp = -1; }

void push( int in ) { m\_items[++m\_sp] = in; }

int pop() { return m\_items[m\_sp--]; }

bool is\_empty() { return (m\_sp == -1); }

Memento\* check\_point() {

return new Memento( m\_items, m\_sp+1 );

}

void roll\_back( Memento\* mem ) {

m\_sp = mem->m\_num-1;

for (int i=0; i < mem->m\_num; ++i)

m\_items[i] = mem->m\_items[i];

}

friend ostream& operator<< ( ostream& os, const Stack& s ) {

string buf( "[ " );

for (int i=0; i < s.m\_sp+1; i++) {

buf += s.m\_items[i]+48;

buf += ' ';

}

buf += ']';

return os << buf;

}

};

int main( void ) {

Stack s;

for (int i=0; i < 5; i++)

s.push( i );

cout << "stack is " << s << '\n';

Memento\* first = s.check\_point();

for (int i=5; i < 10; i++)

s.push( i );

cout << "stack is " << s << '\n';

Memento\* second = s.check\_point();

cout << "popping stack: ";

while ( ! s.is\_empty())

cout << s.pop() << ' ';

cout << '\n';

cout << "stack is " << s << '\n';

s.roll\_back( second );

cout << "second is " << s << '\n';

s.roll\_back( first );

cout << "first is " << s << '\n';

cout << "popping stack: ";

while ( ! s.is\_empty())

cout << s.pop() << ' ';

cout << '\n';

delete first; delete second;

}

// 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

// stack is [ ]

// second is [ 0 1 2 3 4 5 6 7 8 9 ]

// first is [ 0 1 2 3 4 ]

// popping stack: 4 3 2 1 0

**Observer:**

|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Observer | Notify all observers when one changes its state….Dependents, Publish-Subscribe | Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. |

**Example:**

Both a spreadsheet/table object and bar/pie chart object can depict information in the same application data object using different presentations. The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need. But they behave as though they do. When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa. The Observer pattern describes how to establish these relationships. The key objects in this pattern are subject and observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.

**Related Patterns:**

Mediator : By encapsulating complex update semantics, the ChangeManager acts as mediator between subjects and observers.  
Singleton : The ChangeManager may use the Singleton pattern to make it unique and globally accessible.

The Observer Pattern is often applied in programming graphical user interfaces. Interesting

matters in this area are data dependencies between several output windows. An example: a

budget is calculated for the next period using three different windows. One is displaying the

actual distribution in a pie - chart while another displays the budget at the same time in a bar -

chart. Changing the budget both diagrams are expected to show the new distribution at once.

Therefore it is compelling to include a mechanism to inform the diagram displaying components

of the program for changes in the budget. Also elements as buttons or lists have to inform the

program when the user changes their condition.

An Observer Pattern creates a relationship between a subject and an observer. An Object is

called a subject, when changes in its condition are interesting for other objects. Observer objects

instead are depending on the condition of the subjects - following the subject they have to

change their own condition. Observer Patterns offer certain characteristics to handle the

communication

- An observer Interface defines an update() method. The update() method gets called

for every concrete observer in case of a change in the condition of the observed

subject.

- A class subject defines methods for the obervers to register. In case of a change in

the condition of the subject it calls the update() method of all registered subjects.

- A concrete subject calls for each change in its condition a fireUpdate() method.

Subsequent all registered observers can react to the change and modify their own

condition.

- A concrete observer implements the observer interface and registers for all

required subjects. In its method update() it implements the code needed to execute

for a change in the condition of one of the subjects.

At first the Observer Interface is now not really an Interface as C++ does not support the

same interface mechanism as Java does. Therefore it is an abstract class. Again this is not

implemented by a keyword as in Java. The compiler treats it as an abstract class because the

only defined method update() is pure virtual. The concrete observer class Printer inherits from

the Observer class. This is compelling as the Set in the Subject class (which is supposed to store

all observers) only stores pointers to observer classes - and not to a concrete Printer class. Of

course it’s undeniable that in big software systems there exists a big variety on different

concrete observer classes - and therefore the Set must be defined for pointers to the more

general abstract observer class.

As mentioned the Subject class support a Set to store concrete observers and not an Array.

This is again a difference to the example given in Java. This design benefits as the Set is not

limited to only five observers and therefore it is more flexible.

The Set stores pointers to the concrete observer objects. Managing it in this way it is easy

to inform the observers in case of a change in the concrete subjects condition. As all collections

also the Set provides an iterator to go through the complete Set for calling the update() method

for all registered servers.

Example:

#ifndef OBSERVER\_H

#define OBSERVER\_H

class Observer{

public:

virtual void update() = 0;

};

#endif // OBSERVER\_H

#include "Observer.h"

#include <set>

#include <iostream>

using namespace std;

class Subject {

std::set<Observer\*> observers;

public:

virtual void attach(Observer& observer) {

observers.insert(&observer);

}

virtual void detach(Observer& observer) {

observers.erase(&observer);

}

virtual void fireUpdate() {

set<Observer\*>::iterator i;

i = observers.begin();

for(;i != observers.end(); i++) {

(\*i)->update();

}

}

};

class Counter {

Subject subject;

public:

int cnt;

void attach(Observer& observer) {

subject.attach(observer);

}

void detach(Observer& observer) {

subject.detach(observer);

}

void increase()

{

cnt++;

if (cnt % 3 == 0) {

subject.fireUpdate();

}

}

};

class Printer : public Observer {

public:

void update() {

cout << "Divisable by 3" << endl;

}

};

int main() {

Printer printer;

Counter counter;

counter.attach(printer);

counter.cnt = 0;

while (counter.cnt < 10) {

counter.increase();

cout << counter.cnt << endl;

}

}

The Observer Pattern is often used in Java, as the communication between graphical

dialogue elements and their application is based on this idea. Although it is slightly extended

and different. The observer is called Listener. It’s also usual that a listener registers to more than

one subject and that it provides more than one update() method. The listener-concept of Java is

called Delegation Based Event Handling.

**State:**

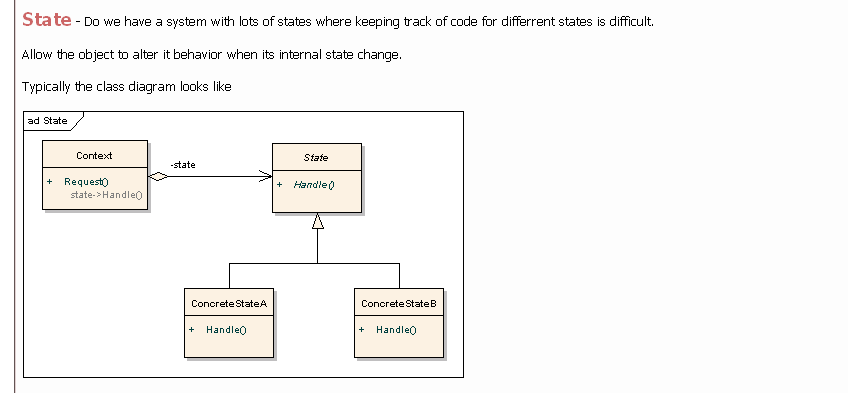
|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| State | Change behavior as per state…..Objects for States | Allow an object to alter its behavior when its internal state changes. The object will appear to change its class. |

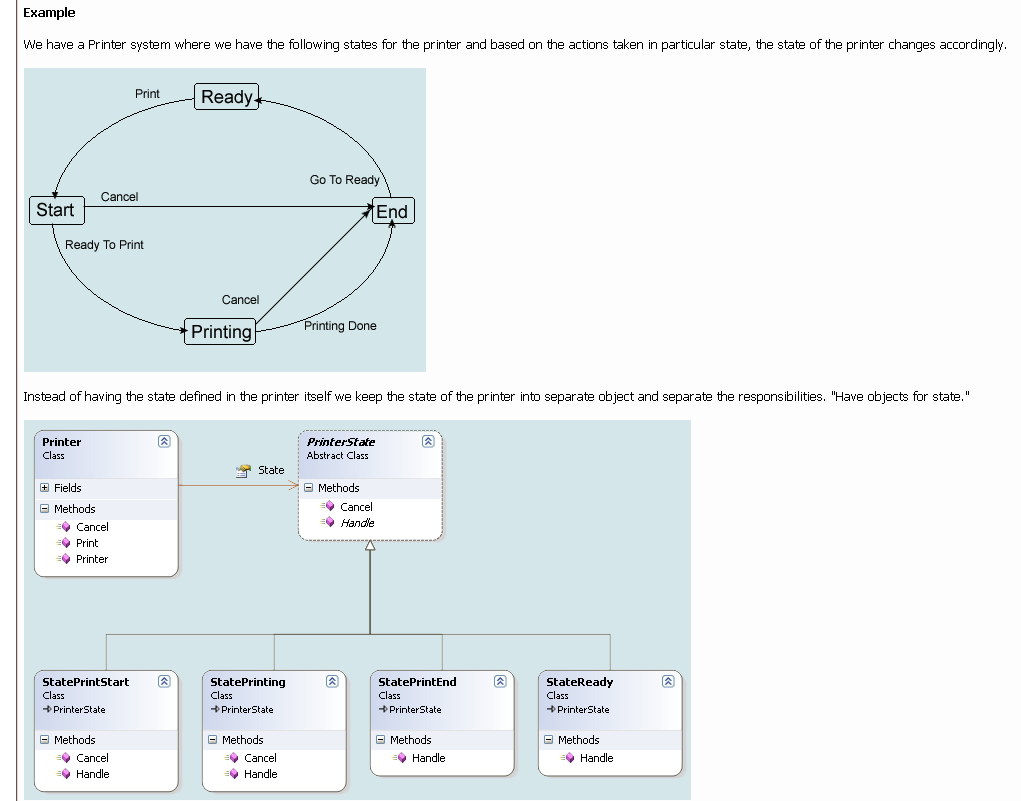
**Example:**

Consider a class TCPConnection that represents a network connection. A TCPConnection object can be in one of several different states: Established, Listening, Closed. When a TCPConnection object receives requests from other objects, it responds differently depending on its current state. For example, the effect of an Open request depends on whether the connection is in its Closed state or its Established state. The State pattern describes how TCPConnection can exhibit different behavior in each state.

**Related Patterns:**

The Flyweight pattern explains when and how State objects can be shared.  
State objects are often Singletons.





// There are alot of applications which need to maintain different states on

// different contexts. Mainly, it comes in the matter of 3D-Applications and

// devices. Without using state pattern, normally we are handling the state as

// follows

class MyDevice

{

int m\_nCurrentState;

public:

MyDevice()

{

m\_nCurrentState= STATE\_OFF;

}

// Function handling the state changes

void HandleState();

};

void MyDevice::HandleState()

{

/\* changing state depends on the current state \*/

if(STATE\_OFF == m\_nCurrentState)

{

m\_nCurrentState = STATE\_ON;

}

else if(STATE\_ON == m\_nCurrentState)

{

m\_nCurrentState = STATE\_ACQUIRING;

}

else if(STATE\_ACQUIRING == m\_nCurrentState)

{

m\_nCurrentState = STATE\_SAVE;

}

....

....

// We can add more states here

}

// What the problem with the above code is, usually the state changing and other

// activities will be centralized, so that the main function doing this activities

// will get complex and lengthy if there are many states are to be handled.

// This function will also get complex in the matter of maintainability.

// For handling few number of states, this aproach is most suitable and easy to handle.

// The state pattern comes when complexity matters.

// The concept behind the state pattern:

// In state pattern, an object can have different states each state knows what its

// next state is. By using this architecture we can add or remove a state at ease.

// This architecture mainly consists of 3 types of classes.

// 1. Context Class

// This class is the one which has the states. The application will be using (interfering with)

// this class. The class is responsible to maintain current state.

// 2. State Class

// This is an abstract base class which can hold the different states (sub classes).

// The context class will be using this class pointer as its member to point the current state.

// 3. Concentrate Class

// This class implements the behavior of the state and behavior of the object.

//

// Example: State Change

// Here I’m taking the Sun as example: A person who is living on earth can view sun in

// different states (even it remains constant). Those are morning, noon, afternoon,

// evening, night. For Simplicity I’m taking 3 states Morning, Evening and Night.

// The Sun is starting (Initial state) with state Morning.

// Morning knows that sun will move to Evening after its time period, Evening knows

// that beautiful Night will come after its turn and finally after a dark Night,

// it knows that Sun will change to a cool Morning. In this architecture each state

// has a next state. When the object receives a command to change its state,

// it will ask the current state, what its next state is and modify itself to

// the new state returned.

// Adding a New State

// Suppose we want to add a new state Noon for Sun. We can do in simple steps as described

// below. Take Noon as example

// 1. Define a new concentrate class (Noon class)

// 2. Define its next state (Evening)

// 3. Give this state as the next state of another state (change Morning’s next state from Evening to Noon).

// State class

class CBaseState

{

public:

// Pure virtual function

virtual CBaseState\* GetNextState() = 0;

// print the string

virtual char\* ToString() = 0;

};

// Concetrate classes

// State Morning

class CMorning : public CBaseState

{

public:

virtual CBaseState\* GetNextState();

virtual char\* ToString();

};

// State Evening

class CEvening : public CBaseState

{

public:

virtual CBaseState\* GetNextState();

virtual char\* ToString();

};

// State night

class CNight: public CBaseState

{

public:

virtual CBaseState\* GetNextState();

virtual char\* ToString();

};

// Context Class

class CSun

{

public:

CSun();

CSun(CBaseState\* pContext /\* Pass Allocated memory \*/);

~CSun();

// Handles the next state

void StateChanged();

char\* GetStateName();

protected:

void DoCleanUp();

// Pointer which holds the current state

// Since this is and base class pointer

// of Concentrate classes, it can holds their objects

CBaseState\* m\_pState;

};

// When State Change Request Comes

// In the above example, the sun will be initialized to any of the states for e.g say morning.

CSun objSun(new CMorning);

// If we need to change its current state to next state, it is possible to by calling

// StateChanged interface proveided by the context class.

// See sample code snippet for initializing and changing the state

CSun objSun(new CMorning);

printf("\n\nSun Says Good %s !!!",objSun.GetStateName());

// inform that state has been changed

objSun.StateChanged();

printf("\n\nSun Says Good %s !!!",objSun.GetStateName());

// inform that state has been changed

objSun.StateChanged();

printf("\n\nSun Says Good %s !!!",objSun.GetStateName());

// inform that state has been changed

objSun.StateChanged();

printf("\n\nSun Says Good %s !!!",objSun.GetStateName());

// Handles the state change

void CSun::StateChanged()

{

if (m\_pState)

{

// Getting Next State

CBaseState\* pState = m\_pState->GetNextState();

// de allocates the memory

delete m\_pState;

m\_pState = pState;

}

}

// Defining Next State

// On calling the GetNextState virtual function depends on the object

// each object will return its next state.

// Morning object’s next state

CBaseState\* CMorning::GetNextState()

{

return new CEvening;

}

// Evening object’s next state Collapse Copy Code

CBaseState\* CEvening::GetNextState()

{

return new CNight;

}

// Night object’s next state Collapse Copy Code

CBaseState\* CNight::GetNextState()

{

return new CMorning;

}

// The above described code is a very basic level implementation of state Pattern.

// You can add more interfaces for your class to manage the state (e.g. setting and getting state)

// and improve the code with a good memory handling strategy.

**Strategy:**

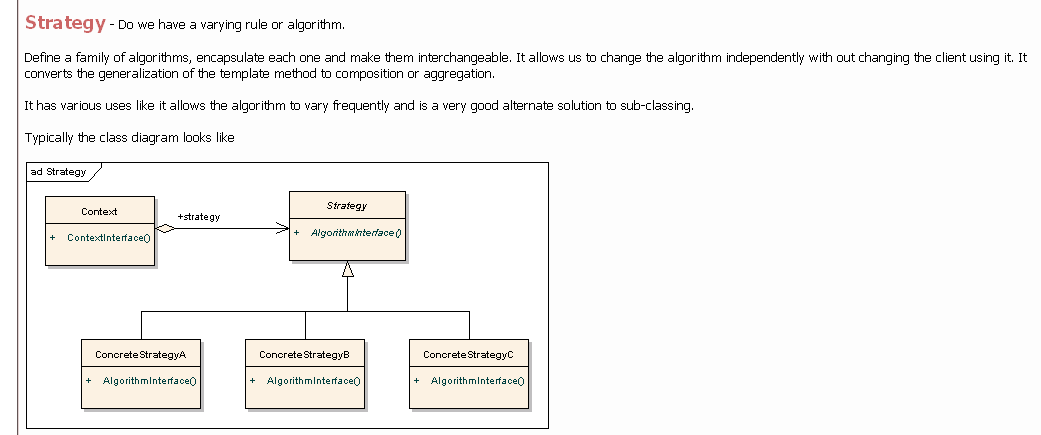
|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Strategy | Formatting algorithm for document editor….Policy | Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. |

**Example:**

Strategy to allow different formatting algorithms. In case study,...Formatting. How does Lexi actually arrange text and graphics into lines and columns? What objects are responsible for carrying out different formatting policies? How do these policies interact with the document's internal representation?   
Composite pattern represents the document's physical structure. Next, we need to figure out how to construct a particular physical structure, one that corresponds to a properly formatted document. Representation and formatting are distinct: The ability to capture the document's physical structure doesn't tell us how to arrive at a particular structure. This responsibility rests mostly on Lexi. It must break text into lines, lines into columns, and so on, taking into account the user's higher-level desires. For example, the user might want to vary margin widths, indentation, and tabulation; single or double space; and probably many other formatting constraints.6 Lexi's formatting algorithm must take all of these into account.

**Related Patterns:**

Flyweight : Strategy objects often make good flyweights.



In computer programming, the strategy pattern (also known as the policy

pattern) is a particular software design pattern, whereby algorithms can be

selected at runtime.

The strategy pattern is useful for situations where it is necessary to

dynamically swap the algorithms used in an application. The strategy pattern is

intended to provide a means to define a family of algorithms, encapsulate each

one as an object, and make them interchangeable. The strategy pattern lets the

algorithms vary independently from clients that use them.

#include <iostream>

using namespace std;

class StrategyInterface

{

public:

virtual void execute() = 0;

};

class ConcreteStrategyA: public StrategyInterface

{

public:

virtual void execute()

{

cout << "Called ConcreteStrategyA execute method" << endl;

}

};

class ConcreteStrategyB: public StrategyInterface

{

public:

virtual void execute()

{

cout << "Called ConcreteStrategyB execute method" << endl;

}

};

class ConcreteStrategyC: public StrategyInterface

{

public:

virtual void execute()

{

cout << "Called ConcreteStrategyC execute method" << endl;

}

};

class Context

{

private:

StrategyInterface \*\_strategy;

public:

Context(StrategyInterface \*strategy):\_strategy(strategy)

{

}

void set\_strategy(StrategyInterface \*strategy)

{

\_strategy = strategy;

}

void execute()

{

\_strategy->execute();

}

};

int main(int argc, char \*argv[])

{

ConcreteStrategyA concreteStrategyA;

ConcreteStrategyB concreteStrategyB;

ConcreteStrategyC concreteStrategyC;

Context contextA(&concreteStrategyA);

Context contextB(&concreteStrategyB);

Context contextC(&concreteStrategyC);

contextA.execute();

contextB.execute();

contextC.execute();

contextA.set\_strategy(&concreteStrategyB);

contextA.execute();

contextA.set\_strategy(&concreteStrategyC);

contextA.execute();

return 0;

}

**Template:**

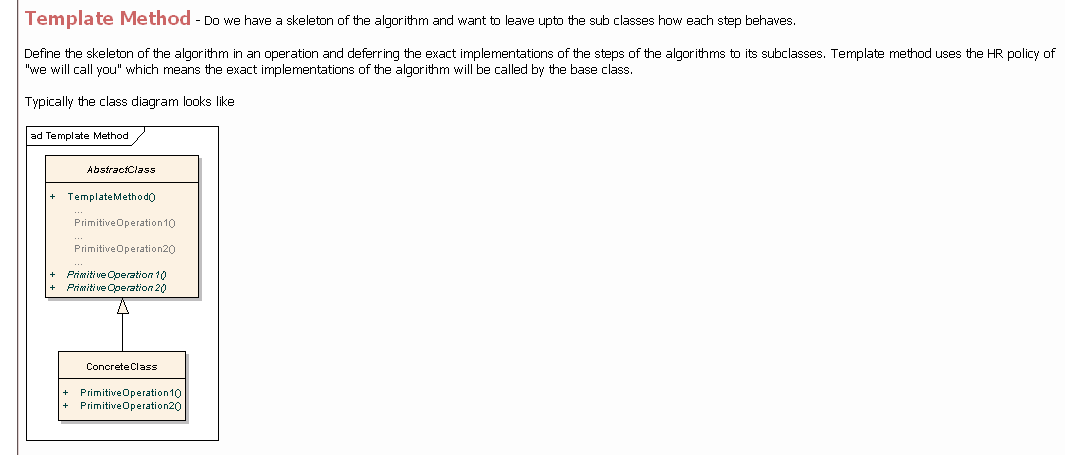
|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Template Method | skeleton of alogorithm… defer some steps to redefine by subclasses | Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. |

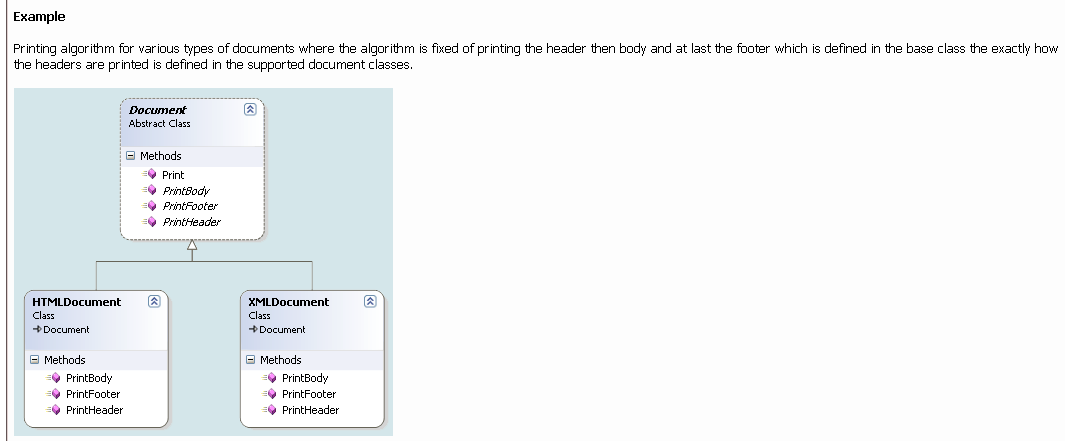
**Example:**

Consider an application framework that provides Application and Document classes. The Application class is responsible for opening existing documents stored in an external format, such as a file. A Document object represents the information in a document once it's read from the file.  
Applications built with the framework can subclass Application and Document to suit specific needs. For example, a drawing application defines DrawApplication and DrawDocument subclasses; a spreadsheet application defines SpreadsheetApplication and SpreadsheetDocument subclasses. The abstract Application class defines the algorithm for opening and reading a document in its OpenDocument operation. OpenDocument defines each step for opening a document. It checks if the document can be opened, creates the application-specific Document object, adds it to its set of documents, and reads the Document from a file.  
We call OpenDocument a template method. A template method defines an algorithm in terms of abstract operations that subclasses override to provide concrete behavior.

**Related Patterns:**

Factory Methods (107) are often called by template methods. In the Motivation example, the factory method DoCreateDocument is called by the template method OpenDocument.  
Strategy : Template methods use inheritance to vary part of an algorithm. Strategies use delegation to vary the entire algorithm.





// The common implementation has been moved to

// an abstract base class, and a "placeholder"

// has been defined to encapsulate the embedded

// comparison.

// All that remains for the SortUp and SortDown

// classes is to implement that placeholder.

class AbstractSort { ////// Shell sort //////

public:

void sort( 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 (needSwap( v[j], v[j+g] ))

doSwap(v[j], v[j+g]);

}

private:

virtual int needSwap(int,int) = 0;

void doSwap(int& a,int& b) {

int t = a; a = b; b = t;

}

};

class SortUp : public AbstractSort {

/\* virtual \*/ int needSwap(int a, int b) {

return (a > b);

}

};

class SortDown : public AbstractSort {

/\* virtual \*/ int needSwap(int a, int b) {

return (a < b);

}

};

int main( void ) {

const int NUM = 10;

int array[NUM];

srand( (unsigned) time(0) );

for (int i=0; i < NUM; i++) {

array[i] = rand() % 10 + 1;

cout << array[i] << ' ';

}

cout << '\n';

AbstractSort\* sortObjects[] = {

new SortUp, new SortDown };

sortObjects[0]->sort( array, NUM );

for (int u=0; u < NUM; u++)

cout << array[u] << ' ';

cout << '\n';

sortObjects[1]->sort( array, NUM );

for (int d=0; d < NUM; d++)

cout << array[d] << ' ';

cout << '\n';

system( "pause" );

}

// 1 6 6 2 10 9 4 10 6 4

// 1 2 4 4 6 6 6 9 10 10

// 10 10 9 6 6 6 4 4 2 1

**Visitor:**

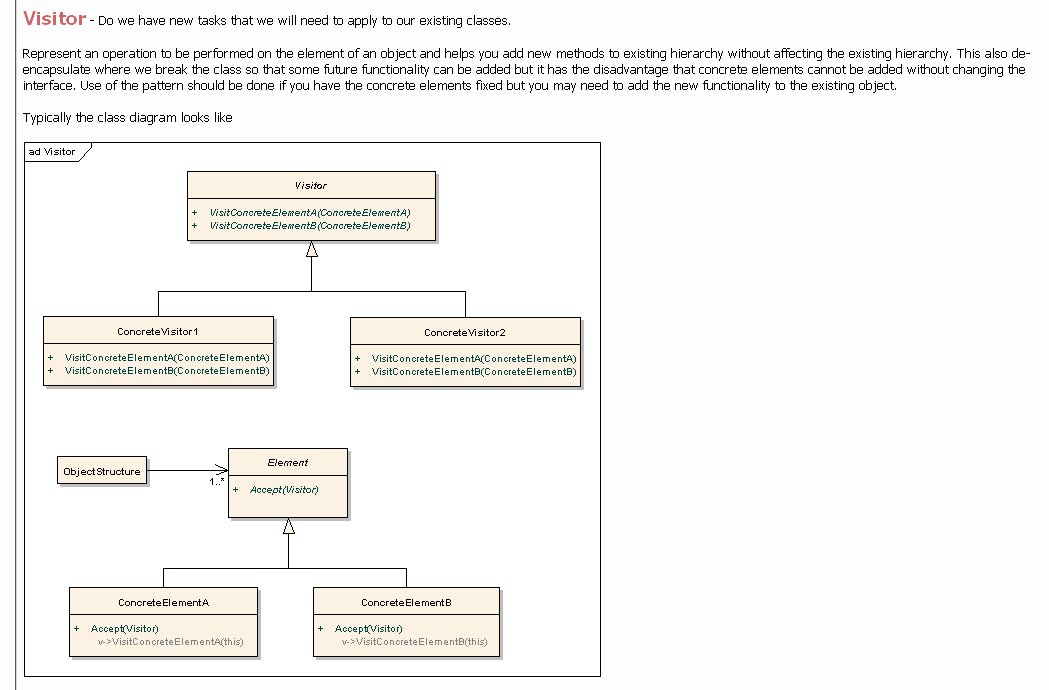
|  |  |  |
| --- | --- | --- |
| Pattern | Also known as | Intent |
| Visitor | spell check and hyphenation in document editor | Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates. |

**Example:**

Visitor for allowing an open-ended number of analytical capabilities without complicating the document structure's implementation. In case study,..Spelling checking and hyphenation. How does Lexi support analytical operations such as checking for misspelled words and determining hyphenation points? How can we minimize the number of classes we have to modify to add a new analytical operation?  
Iterator pattern defines a way of traversing the glyph structure, we need to check the spelling and do the hyphenation. Both analyses involve accumulating information during the traversal. analysis and traversal should be separate. we need to encapsulate the analysis in a separate object. The iterator would "carry" the instance to each glyph in the structure. The analysis object could then perform a piece of the analysis at each point in the traversal. The analyzer accumulates information of interest (characters in this case) as the traversal proceeds.

**Related Patterns:**

Composite : Visitors can be used to apply an operation over an object structure defined by the Composite pattern.  
Interpreter : Visitor may be applied to do the interpretation.



- 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.

// Before

// BEFORE - The interface for "operations" are

// specified in the Color base class and imple-

// mented in the Color derived classes.

class Color {

public:

virtual void count() = 0;

virtual void call() = 0;

static void report\_num() {

cout << "Reds " << s\_num\_red

<< ", Blus " << s\_num\_blu << '\n';

}

protected:

static int s\_num\_red, s\_num\_blu;

};

int Color::s\_num\_red = 0;

int Color::s\_num\_blu = 0;

class Red : public Color {

public:

void count() { ++s\_num\_red; }

void call() { eye(); }

void eye() { cout << "Red::eye\n"; }

};

class Blu : public Color {

public:

void count() { ++s\_num\_blu; }

void call() { sky(); }

void sky() { cout << "Blu::sky\n"; }

};

int 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::report\_num();

}

// Red::eye

// Blu::sky

// Blu::sky

// Red::eye

// Red::eye

// Reds 3, Blus 2

//

// After

// AFTER - The Color hierarchy specifies a

// single "accept()" method, and then the pre-

// vious "count()" and "call()" methods are

// implemented as Visitor derived classes.

// When accept() is called on a Color object,

// that is the first dispatch. When visit()

// is called on a Visitor object, that is the

// second dispatch; and the "right thing" can

// be done based on the type of both objects.

class Color {

public:

virtual void accept( class Visitor\* ) = 0;

};

class Red : public Color {

public:

/\*virtual\*/ void accept( Visitor\* );

void eye() { cout << "Red::eye\n"; }

};

class Blu : public Color {

public:

/\*virtual\*/ void accept( Visitor\* );

void sky() { cout << "Blu::sky\n"; }

};

class Visitor {

public:

virtual void visit( Red\* ) = 0;

virtual void visit( Blu\* ) = 0;

};

class CountVisitor : public Visitor {

public:

CountVisitor() { m\_num\_red = m\_num\_blu = 0; }

/\*virtual\*/ void visit( Red\* ) { ++m\_num\_red; }

/\*virtual\*/ void visit( Blu\* ) { ++m\_num\_blu; }

void report\_num() {

cout << "Reds " << m\_num\_red

<< ", Blus " << m\_num\_blu << '\n';

}

private:

int m\_num\_red, m\_num\_blu;

};

class CallVisitor : 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 ); }

int main( void ) {

Color\* set[] = { new Red, new Blu, new Blu,

new Red, new Red, 0 };

CountVisitor count\_operation;

CallVisitor call\_operation;

for (int i=0; set[i]; i++) {

set[i]->accept( &count\_operation );

set[i]->accept( &call\_operation );

}

count\_operation.report\_num();

}

// Red::eye

// Blu::sky

// Blu::sky

// Red::eye

// Red::eye

// Reds 3, Blus 2

-----------------------------------------------------------------------------------------

Some Notes:

-----------------------------------------------------------------------------------------

Difference between Factory method and abstract factory.

Factory method takes care of one product, whereas the abstract factory pattern provides a way to encapsulate a family of products.

Use factory method to create Abstact Factory Subclass which creates a concrete object/product.

------------------------------------------------------------------------------------------

Use singleton to create a prototype and from prototype a concrete product can be created using prototype pattern.

------------------------------------------------------------------------------------------

bird flyingBirds, nonFlyingBirds, sparrow, penguin->fly() design problem

------------------------------------------------------------------------------------------

dll 1, 2, 3, exe. function call in dll2 to function in dll3. build seq.... how to call fun without link error? - use of virtual function. chain of responsibility pattern. bridge between two dlls

------------------------------------------------------------------------------------------