Definition

*The Proxy Pattern provides a surrogate or placeholder for another object to control access to it.*

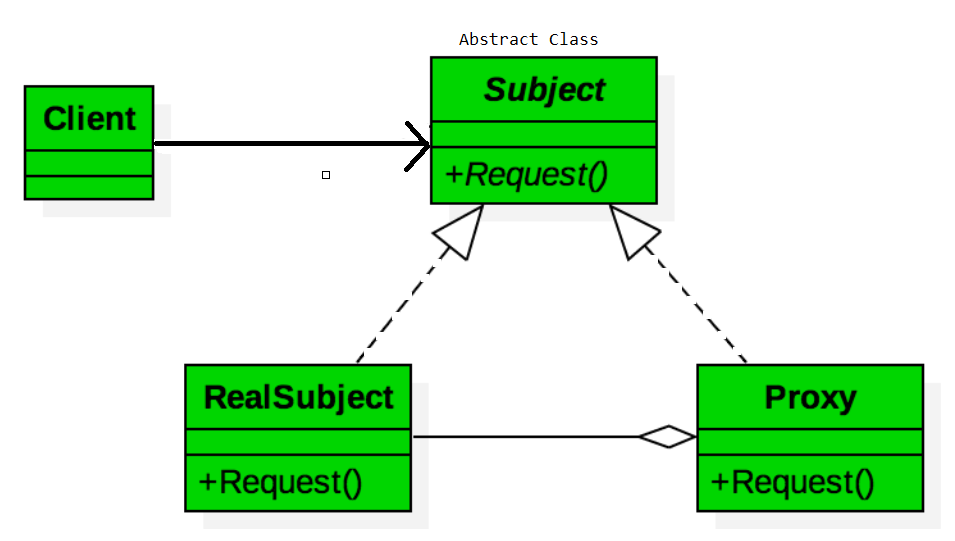
You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.

For example, if we need to use only a few methods of some costly objects. We'll initialize those objects when we need them entirely. Until that point we can use some light objects exposing the same interface as the heavy objects.

These light objects are called proxies and they will instantiate those heavy objects when they are really needed.

Consider for example, an image viewer program.

An image viewer program must be able to list and display high resolution photo objects that are in a folder, but how often do someone open a folder and view all the images inside. Sometimes you will be looking for a particular photo, sometimes you will only want to see an image name. The image viewer must be able to list all photo objects, but the photo objects must not be loaded into memory until they are required to be rendered.



The above class diagram represents the Proxy Pattern in its simplest form. Our surrogate or place holder would be our ***Proxy*** object that controls the access to another object ***RealSubject***. Each implement the same interface, ***Subject***. This allows the client to implement to an interface and have no knowledge that it is interacting with a ***Proxy*** versus the ***RealSubject***.

The participants classes in the proxy pattern are:

* **Subject** - Interface implemented by the RealSubject and representing its services. The interface must be implemented by the proxy as well so that the proxy can be used in any location where the RealSubject can be used.
* **Proxy**
* Maintains a reference that allows the Proxy to access the RealSubject.
* Implements the same interface implemented by the RealSubject so that the Proxy can be substituted for the RealSubject.
* Controls access to the RealSubject and may be responsible for its creation and deletion.
* Other responsibilities depend on the kind of proxy.
* **RealSubject** - the real object that the proxy represents.

Types of Proxies

There are a few different types of proxies. Each control access to different types of resources.

1. **Remote Proxy** – Our RealSubject is a remote object whose access is controlled by a Proxy object. it hides the fact that the *RealSubject* lives within another process.
2. **Virtual Proxy** – Our RealSubject or a resource that is part of our RealSubject is expensive to create. The Proxy object, in this case, would defer the creation until it was absolutely needed.
3. **Protection Proxy** – Our RealSubject may require access protection of some kind. The Proxy would provide that protection based on access rights.
4. **Smart Reference** – Our Proxy would provide additional behavior such as reference counts of the RealSubject pointer. This is commonly known as a smart pointer. Additionally, the Proxy may provide a locking mechanism to protect the RealSubject from being changed by other objects while an access is already in progress. Lastly, the Proxy could be responsible for creating the RealSubject when it is first needed.

Related Patterns

**Adapter Design Pattern** - The adapter implements a different interface to the object it adapts where a proxy implements the same interface as its subject.

**Decorator Design Pattern** - A decorator implementation can be the same as the proxy however a decorator adds responsibilities to an object while a proxy controls access to it.

Protection proxy Example**:**

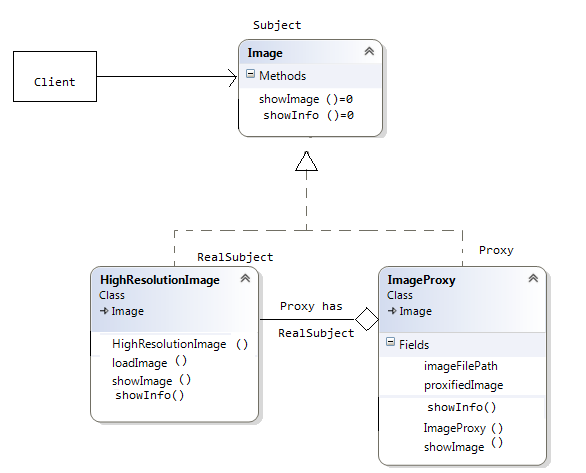
Controls access to the original object

Java Remote Method Invocation (RMI)

In java RMI an object on one machine (executing in one JVM) called a client can invoke methods on an object in another machine (another JVM) the second object is called a remote object. The proxy (also called a stub) resides on the client machine and the client invokes the proxy in as if it is invoking the object itself (remember that the proxy implements the same interface that RealSubject implements). The proxy itself will handle communication to the remote object, invoke the method on that remote object, and would return the result if any to the client. The proxy in this case is a Remote proxy.

Virtual Proxy Example

Consider an image viewer program that lists and displays high resolution photos. The program must show a list of all photos however it does not need to display the actual photo until the user selects an image item from a list.



The code below shows the Image interface representing the Subject. The interface has a single method showImage() that the Concrete Images must implement to render an image to screen.

class Image {

public:

virtual void showImage() = 0;

virtual void showInfo() = 0;

};

The code below displays the ***RealSubject*** Implementation, which is the concrete and heavyweight implementation of the image interface. The High resolution image, loads a high resolution image from disk, and renders it to screen when showImage() is called.

class HighResolutionImage : public Image {

public:

HighResolutionImage(std::string imageFilePath) {

loadImage(imageFilePath);

}

void loadImage(std::string imageFilePath) {

// load Image from disk into memory

// this is heavy and costly operation

}

//@Override

void showImage() {

// Actual Image rendering logic

}

void showInfo() {

//It is low cost operation.

//Show ISO,F-stop,Exposer setting,file Name,size etc.

}

};

The code below shows the ***Proxy*** implementation, the image proxy is a virtual proxy that creates and loads the actual image object on demand, thus saving the cost of loading an image into memory until it needs to be rendered:

class ImageProxy : public Image {

std::string imageFilePath; //Private Proxy data

Image \*proxifiedImage; //Reference to RealSubject

public:

ImageProxy(std::string imageFilePath) {

this->imageFilePath = imageFilePath;

}

void showImage() {

// create the Image Object only when the image is required to be shown

proxifiedImage = new HighResolutionImage(imageFilePath);

// now call showImage on realSubject

proxifiedImage->showImage();

}

void showInfo() {

//It is low cost operation.

//Show ISO,F-stop,Exposer setting,file Name,size etc.

}

};

The code below illustrates a sample image viewer program; the program simply loads three images, and renders only one image, once using the proxy pattern, and another time directly. Note that when using the proxy pattern, although three images have been loaded, the High resolution image is not loaded into memory until it needs to be rendered, while in the part not using the proxy, the three images are loaded into memory although one of them is actually rendered.

int main() {

//assuming that the user selects a folder that has 3 images create the 3 images

Image \*highResolutionImage1 = new ImageProxy("sample/veryHighResPhoto1.jpeg");

Image \*highResolutionImage2 = new ImageProxy("sample/veryHighResPhoto2.jpeg");

Image \*highResolutionImage3 = new ImageProxy("sample/veryHighResPhoto3.jpeg");

To show the image information, donot need to load image into the memory.

It is light weight operation, for that not neeed to create bulky object.

highResolutionImage1->showInfo();

Assume that the user clicks on Image one item in a list this would cause the

Program to call showImage() for that image only note that in this case only image

One was loaded into memory

highResolutionImage1->showImage();

//Consider using the high resolution image objects directly

Image \*highResolutionImageNoProxy1 = new

HighResolutionImage("sample/veryHighResPhoto1.jpeg");

Image \*highResolutionImageNoProxy2 = new

HighResolutionImage("sample/veryHighResPhoto2.jpeg");

Image \*highResolutionImageBoProxy3 = new

HighResolutionImage("sample/veryHighResPhoto3.jpeg");

//Assume that the user selects image two item from images list

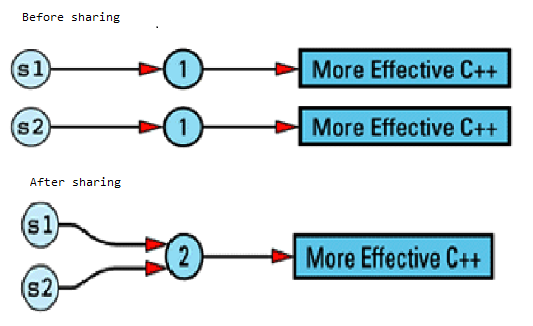
highResolutionImageNoProxy2->showImage();

Note that in this case all images have been loaded into memory and not all have

been actually displayed this is a waste of memory resources

}

Reference Counting. (Smart Reference Proxy)



class RCString

{

private:

struct StringValue {

int refCount;

char \*data;

StringValue(const char \*initValue);

~StringValue();

};

StringValue \*value; // value of this String

public:

RCString(const char \*initValue = "");

RCString(const RCString& rhs);

RCString& operator=(const RCString& rhs);

char& operator[](int index)const; //For Const String

char& operator[](int index); //For non-Const String.

~RCString();

void showStr()const {

cout <<"Data:: "<<value->data<<", Address::"<< &(value->data)<<endl;

cout << "Ref cnt:: " << value->refCount<<endl;

cout << "---------------\n";

}

};

//Implementation of the const version of this function is straightforward, because it's a //read-only operation; the value of the string can't be affected

char& RCString::operator[](int index)const

{

if ((strlen(value->data) - 1) > index)

return value->data[index];

///else throw ArrayOutOfBoundExeception

}

//Non-const version, Since we don’t know whether it is read or write, so we are creating //new set of stringValue therefore it cannot create problem while write operation happening.

char& RCString::operator[](int index) //throw ArrayOutOfBoundExeception,Implement leter..

{

// if we're sharing a value with other String objects, break off a separate copy

// of the value for ourselves

if (value->refCount > 0)

{

// decrement current value's refCount, because we won't be using that value

// any more

value->refCount--;

value = new StringValue(value->data); // make a copy of the value for

//ourselves.

}

// return a reference to a character inside our unshared StringValue object

return value->data[index];

}

RCString& RCString::operator=(const RCString& rhs)

{

if (this->value == rhs.value)

return \*this;

this->~RCString();

this->value = rhs.value;

this->value->refCount++;

return \*this;

}

RCString::~RCString()

{

if ((--value->refCount) == 0) {

value->~StringValue();

}

}

RCString::RCString(const char \*initValue): value(new StringValue(initValue))

{

}

RCString::RCString(const RCString& rhs):value(rhs.value) {

++value->refCount;

}

RCString::StringValue::StringValue(const char \*initValue):refCount(1) {

data = new char[strlen(initValue) + 1];

strcpy(data, initValue);

}

RCString::StringValue::~StringValue(){

delete[] data;

}

int main() {

Problem with Write operation-When we modify a String's value, we have to be careful to avoid modifying the value of other String objects. Because we all share same value among classes and unfortunately, there is no way for C++ compilers to tell us whether a particular use of operator[] is for a read or a write.

//Write Operation

RCString str1("RajeevKumarSharma"); //str1->refCount=1

RCString str2(str1); //str1->refCount=2,str2->refCount=2

//Both have same copy of string.

str2[0] = 'X'; //Now we will modifying str2, but It should not be //inmpact str1

str1.showStr(); //str1->refCount=1

str2.showStr(); //str2->refCount=1,It have seperate copy of string //"XajeevKumarSharma"

//Read Operation.

RCString str3("RajeevKumarNayan");

RCString str4(str3);

cout << str3[0] << endl; //Read, Unfortunately in operator[] call, we canot determine

//whether it is called by read or write operation.

str3.showStr(); //str3->refCount=1

str4.showStr(); //str4->refCount=1

//Still having seperate copy of string "RajeevKumarNayan" which //is ellogical.

const RCString str5("ThisIsConstString");

str5.showStr();

char ch1 = str5[3]; //Non Const RCString Read.

str5.showStr(); //No address change after read, because of const

RCString str6("ThisIsNonConstString")

str6.showStr();

char ch2 = str6[1]; //String changed while reading.

str6.showStr(); //Address changed.

getchar();

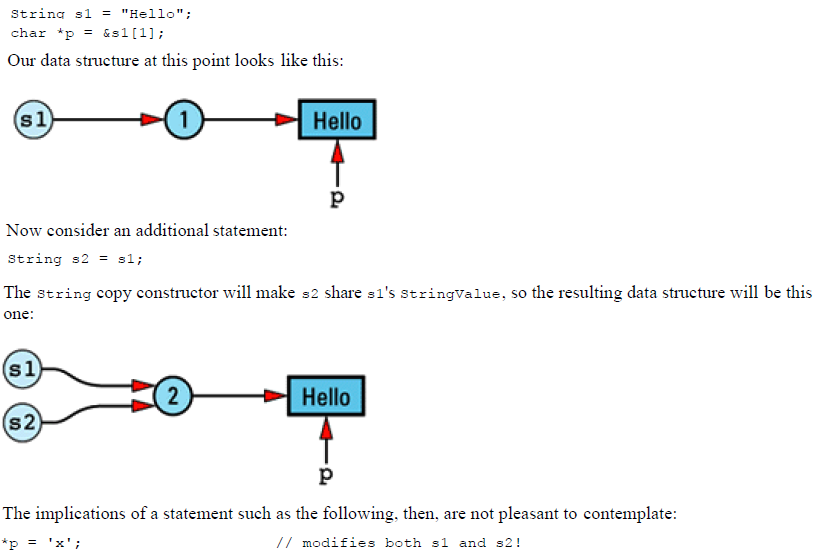
return 0;

}

Pointers, References, and Copy-on-Write:

This idea — that of sharing a value with other objects until we have to write on our own copy of the value — has a long and distinguished history in Computer Science, especially in operating systems, where processes are routinely allowed to share pages until they want to modify data on their own copy of a page. The technique is common enough to have a name: ***copy-on-write***.

Copy-on-write allows us to preserve both efficiency and correctness.



There is no way the String copy constructor can detect this problem, because it has no way to know that a pointer into s1's ***StringValue*** object exists. And this problem isn't limited to pointers, it would exist in previous example when we call to ***RCString*** non-const operator[].