Proxy Design Pattern in Modern C++

Reading Time: 5 minutes

In software engineering, Structural Design Patterns deal with the relationship between objects i.e. how objects/classes interact or build a relationship in a manner suitable to the situation. The Structural Design Patterns simplify the structure by identifying relationships. In this article of the Structural Design Patterns, we're going to take a look at Proxy Design Pattern in C++ which ***dictates the way you access the object***.

If you haven't check out other Structural Design Patterns, then here is the list:

The code snippets you see throughout this series of articles are simplified not sophisticated. So you often see me not using keywords like `override`, `final`, `public`(while inheritance) just to make code compact & consumable(most of the time) in single standard screen size. I also prefer `struct` instead of `class` just to save line by not writing "`public:`" sometimes and also miss virtual destructor, constructor, copy constructor, prefix `std::`, deleting dynamic memory, intentionally. I also consider myself a pragmatic person who wants to convey an idea in the simplest way possible rather than the standard way or using Jargons.

Note:

- If you stumbled here directly, then I would suggest you go through What is design pattern? first, even if it is trivial. I believe it will encourage you to explore more on this topic.
- All of this code you encounter in this series of articles are compiled using C++20(though I have used Modern C++ features up to C++17 in most cases). So if you don't have access to the latest compiler you can use https://wandbox.org/ which has preinstalled boost library as well.

Contents [hide]

Intent

An interface for accessing a particular resource.

- The proxy acts as an interface to a particular resource which may be remote, expensive to construct or require some additional functionality like logging or something else.
- But the key thing about the proxy is that its interface looks just like the interface of the object that you are actually attempting to access. This interface could be a method, overloaded operator or another object of different/local class.

Proxy Design Pattern Examples in C++

- A very sophisticated example of the Proxy Design Pattern in C++ that you're probably using every day already is a smart pointer (like std::unique_ptr, std::shared_ptr, etc.) from the standard library

```
// Ways to access object through pointer
ptr->print();
*ptr = 5;
```

- So let me give you an explanation as to why a smart pointer would be a proxy. Well just by seeing the above code snippet, you can not decide that ptr is a raw pointer or smart pointer.
- Thus smart pointer are proxies as they satisfy both the condition of proxy i.e.

  1. Provide an interface to access the resource.
  2. The interface looks just like the interface of the object.

- There are many different kinds of proxy available like Remote proxy, Virtual proxy, Protection proxy, Communication Proxy. We will see some of them here.

Property Proxy

- As you probably know other programming languages such as C# have this idea of properties. There probably is nothing more than a field plus a getter & setter methods for that field. Let’s suppose that we wanted to get properties in C++ so we have written `Property` class as:

```
template<typename T>
struct Property {
    T   m_value;

    Property(const T initialValue) { * this = initialValue; }
    operator T() { return m_value; }
    T operator = (T newValue) { return m_value = newValue; }
};

struct Creature {
    Property<int32_t>   m_strength{10};
    Property<int32_t>   m_agility{5};
};

int main() {
    Creature creature;
    creature.m_agility = 20;
    cout << creature.m_agility << endl;
    return EXIT_SUCCESS;
}
```

- But seeing above code, you might be wondering that why don’t we just declare `strength` & `agility` as `int32_t`. Now let’s suppose that for some reason you actually wanted ***to “intercept” or “have to log” the assignments as well as the access to these fields***. So you want something which is effective as a `Property` rather than designing the getter & setter method for all the attributes.

Virtual Proxy

- So another type of proxy that you’re bound to encounter at some point is what’s called a Virtual Proxy. Now a Virtual Proxy ***gives you***

the appearance of working with the same object that you're used to working with even though the object might not have even been created.

```
struct Image {
    virtual void draw() = 0;
};

struct Bitmap : Image {
    Bitmap(const string &filename) : m_filename(filename) {
        cout << "Loading image from " << m_filename << endl;
        // Steps to load the image
    }
    void draw() { cout << "Drawing image " << m_filename << endl; }

    string      m_filename;
};

int main() {
    Bitmap img_1{"image_1.png"};
    Bitmap img_2{"image_2.png"};

    (rand() % 2) ? img_1.draw() : img_2.draw();

    return EXIT_SUCCESS;
}
```

- As you can see above, `Bitmap` image is derived from the `Image` interface having polymorphic behaviour as `draw()`. `Bitmap` loads the image eagerly in its constructor.
- At first sight, this seems ok, but the problem with this `Bitmap` is that we don't really need to load the image until the drawing code fires. So there is no point on loading both the images in memory at the time of construction.
- Now let me show you how you can improve the above code without changing `Bitmap`. This kind of technique is quite useful when you are working with a third-party library & wants to

write a wrapper around it for some performance improvements.

```
struct LazyBitmap : Image {
    LazyBitmap(const string &filename) : m_filename(filename) {}
    void draw() {
        if (!m_bmp) m_bmp = make_unique<Bitmap>(m_filename);
        m_bmp->draw();
    }

    unique_ptr<Bitmap>      m_bmp{nullptr};
    string                  m_filename;
};

LazyBitmap img_1{"image_1.png"};
LazyBitmap img_2{"image_2.png"};
```

- As you can see, we are not using `Bitmap` until we need it. Rather we are just caching file name to create `Bitmap` whenever somebody wants to draw an image. So if nobody wants to draw the image there is really no point in loading it from the file.

Communication Proxy(Intuitive Proxy Design Pattern in C++)

- Communication Proxy is by far the most common & intuitive Proxy Design Pattern in C++ you might have come across. A straight forward example of communication proxy is subscript operator overloading. Consider the following example of user-defined type i.e. `arr2D` which works exactly as primitive type 2 dimensional array:

```
template <typename T>
struct arr2D {
    struct proxy {
        proxy(T *arr) : m_arr_1D(arr) {}
        T &operator[](int32_t idx) {
            return m_arr_1D[idx];
        }

        T   *m_arr_1D;
    };

    arr2D::proxy operator[](int32_t idx) {
        return arr2D::proxy(m_arr_2D[idx]);
    }

    T   m_arr_2D[10][10];
};

int main() {
    arr2D<int32_t> arr;
    arr[0][0] = 1;  // Uses the proxy object
    return EXIT_SUCCESS;
}
```

Benefits of Proxy Design Pattern

1. The proxy provides a nice & easy interface for even complex data arrangements.
2. Proxy Design Pattern especially Virtual Proxy also provides performance improvement as we have seen in lazy image loading case above.
3. Property proxy provides the flexibility of logging access to object attributes without the client even knowing.

Summary by FAQs

Is Decorator & Proxy Design Patterns are the same?

They are kind of similar(as both use composition) but used for a different purpose. For example, if you consider the above examples, Proxy usually manages the life cycle & access to objects, whereas the Decorators is a wrapper of the original object having more functionality.

Difference between Adapter, Decorator & Proxy Design Pattern?

- ***Adapter*** provides a different/***compatible interface*** to the wrapped object
- ***Proxy*** provides a somewhat same or ***easy interface***
- ***Decorator*** provides ***enhanced interface***

What are the use cases of Proxy Design Pattern?

- When your objects are resource consuming and you have the most of their time stored on disk, you can use the proxy to act as a placeholder(like we did in lazy image loading above).
- When you want to add access restrictions like object is accessed read-only or making user-based access control before really doing the operations (e.g. if the user is authorised, do the operation, if not, throw an access control exception)

Do you like it☝️? Get such articles directly into the inbox…!?