Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 27: Proxy Design Pattern

Contents

27.1 Introduction

This lecture note introduces the Proxy design pattern. One of the most prevalent implementations of the proxy pattern in C++ is in the implementation of smart pointers. Therefore an overview of smart pointers is presented in the programming preliminaries section before introducing the pattern. Other than using the proxy to implement smart pointers, three other implementation strategies are presented.

27.2 Programming Preliminaries

One of the uses of the Proxy design pattern is to manage smart references. In C++ there has been the notion of a Smart Pointer. As from C++11 this notion has been refined.

A C++ smart pointer is an abstract data type that simulates a pointer while providing additional features intended to reduce bugs caused by the misuse of pointers while retaining efficiency [2]. Smart pointers are used to keep track of the objects they point to for the purpose of memory management. This includes bounds checking and automatic garbage collection.

Many libraries exist that implement a type of smart pointer. The library under consideration in this section is the STL for C++11. The standard for C++11 was accepted in August 2011 and is being implemented in C++ compilers. In this standard, there are 3 kinds of smart pointers, `unique_ptr`, `share_ptr` and `weak_ptr`. `unique_ptr` can be replaced with `auto_ptr` as defined in the C++ standard prior to C++11. As with `auto_ptr`, it is defined in the header `<memory>`. The latter two are based on the implementation of smart pointers in the Boost library.

auto_ptr - `auto_ptr` is not implemented in C++11. It is however discussed here for the sake of completeness. Up till C++11 it was the only smart pointer implementation available in C++. In the way `auto_ptr` is implemented, the copy constructor and assignment operators do not copy the stored pointer. They copy the pointer, leaving the copied or assigned object empty. This strategy effectively transfers ownership of the pointer. It however does not provide a solution when copy semantics are required.

```
auto_ptr<int> value(new int(10));

cout<<value.get()<<endl; // access to the pointer
cout<<*value<<endl;      // access to the value of the pointer
```

unique_ptr - unique ownership, move constructible and move assignable. Changing the example given for `auto_ptr` can be adapted to make use of C++11 `unique_ptr` by replacing all references to `auto_ptr` with references to `unique_ptr`. Note, it may be necessary when compiling and linking to inform the compiler that C++11 is required by specifying the correct flags for your compiler if it supports the C++11 standard.

```
unique_ptr<int> value(new int(10));
```

```
cout<<value.get()<<endl; // access to the pointer
cout<<*value<<endl;      // access to the value of the pointer

unique_ptr<int> newValue = move(value); // Transfer ownership

newValue.reset() // deletes the memory
value.reset() // nothing to delete
```

shared_ptr - counted pointer, object is deleted when the use count goes to zero

```
shared_ptr<int>  value(new int(10));
shared_ptr<int>  newValue = value;
                  //newValue and value both own the memory

cout<<value.get()<<endl; // access to the pointer
cout<<*value<<endl;      // access to the value of the pointer

value.reset(); // memory still exists because of dual ownership
newValue.reset(); //last to own the memory, deletes the memory
```

weak_ptr - robust unowned pointer which is managed by a shared pointer

```
shared_ptr<int> value(new int(10));
weak_ptr<int> newValue = value; //value owns the memory
              // newValue holds a reference
```

27.3 Proxy Design Pattern

27.3.1 Identification

Name	Classification	Strategy
Proxy	Structural	Delegation
Intent		
Provide a surrogate or placeholder for another object to control access to it. ([1]:207)		

27.3.2 Problem

The proxy holds off using and therefore possibly also creating the real subject until it is necessary. This means that the cost of accessing the real subject is deferred. The availability of the real subject is therefore on demand. This means that the proxy can be seen as a replacement object where there is a requirement for a more sophisticated reference to an object than just a pointer.

27.3.3 Structure

The structure of the Proxy design pattern is given in Figure 1.

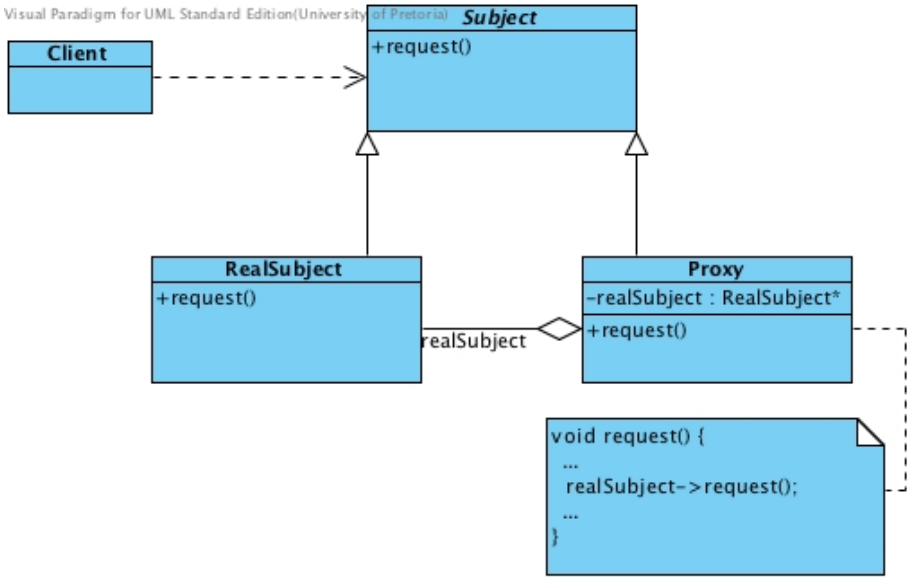

Figure 1: The structure of the Proxy Design Pattern

27.3.4 Participants

Subject

- Defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.

RealSubject

- Defines the real object that is represented by the proxy.

Proxy

- serves as substitute for the real subject
- maintains a reference to the real subject
- controls access to the real subject
- may be responsible for creating and deleting the real subject
- more responsibilities specific to its kind

27.4 Proxy Pattern Explained

The proxy pattern may be applied to a number of situations. The most common situations where a proxy may be applied are given below.

Remote proxy: The remote proxy provides a local representation of an object in a different address space. It therefor is used to hide the fact the the object may be on a different computer. The remote proxy is responsible for:

- encoding a request and its arguments; and

- sending the encoded request to the real subject in a different address space.

Virtual proxy: The virtual proxy provides a local placeholder for an object that is expensive to create and maintain. It is used to postpone access to the expensive object until it is really needed. The virtual proxy is responsible for:

- creating expensive objects on demand; and
- caching information about the real subject so that access to it can be avoided if possible.

Protection proxy: The protection proxy controls access to the real object. It is used to control access rights to the real subject. Different users/objects may have different access rights. The protection proxy checks the access rights of a user/object for the particular request being issued. It may perform additional housekeeping tasks when the object is accessed.

Smart reference: When a proxy is implemented as a smart reference, it replaces a bare pointer and performs additional actions when the object is accessed. The typical uses of a smart pointer are:

Memory management - count the number of references to the real object

Load on demand - load a persistent object into memory on first reference

Safe updating - lock the real object before it is accessed

27.4.1 Related Patterns

Adapter

Adapter and Proxy both provide an interface to access another object. However, the reasons for doing this are different. Adapter a different interface to the object it adapts. Proxy provides the same, or diminished interface to its subject.

Decorator

Decorator and Proxy both describe how to provide a level of indirection to an object and forward requests to it. However, they have a different purpose. Decorator provides a dynamic attach or detachment of an object through recursive composition. The component provides only part of the functionality. One or more decorators provide furnish the rest. Proxy provides a stand-in win it is inconvenient of undesirable to access the subject directly. With the proxy, the subject provides the key functionality. The proxy provides (or refuses) access to this functionality.

Prototype

Prototype and Proxy both offer a solution to a problem related to an object that is expensive to create. However, the solutions are different. Prototype keeps a copy of the object handy and clones it on demand. Proxy creates a stub for the object and created it on demand.

Flyweight

Flyweight and Proxy both apply a smart reference to manage access to an object. However, the purpose of the reference is quite different. Flyweight controls multiple pointers to a shared instance. It can be related back to a C++11 share_ptr. A proxy controls single access to a specific object. This relates to a C++11 unique_ptr.

27.5 Example

One example is presented in two versions. Additional suggestion for other versions of the application of the proxy design pattern is also given after the code for version 2 has been given.

27.5.1 Message Server Example - Version 1

The following code implements a message server. The message server is seen as the RealSubject participant. A proxy participant takes the messages from the client and passes them on to the actual message server. In this example, each proxy message server links to its own actual message server.

```
// Subject
class MessageServer {
public:
  virtual void connect() = 0;
  virtual bool message(string) = 0;
  virtual void disconnect() = 0;
  virtual ~MessageServer() {}
};

//RealSubject
class ActualMessageServer : public MessageServer {
public:
    ActualMessageServer() {
        messages = 0;
    }
    void connect() {
        cout << "Connected..." << endl;
    }
    bool message(string msg) {
        cout << "Message is " << msg << endl; messages++;
        return true;
    }
    void disconnect() {
        cout << "Disconnected, " << messages
             << " message(s) have been received." << endl;
    }
private:
    int messages;
};

//Proxy
class ProxyMessageServer : public MessageServer {
    ActualMessageServer* implementation;
    bool connected;
public:
```

```
    ProxyMessageServer() {
        implementation = new ActualMessageServer();
        connected = false;
    }
    ~ProxyMessageServer() {
        delete implementation;
    }
    void connect() {
        if (!connected) {
            implementation->connect();
            connected = true;
        }
    }
    bool message(string msg) {
        if (connected)
            return implementation->message(msg);
    }

    void disconnect() {
        if (connected) {
            implementation->disconnect();
            connected = false;
        }
    }
};
```

27.5.2 Message Server Example - Version 2

This example adapts the implementation given in Version 1 so that there is only one message server, rather than one per proxy. It will require the proxy to 'register' with the message server when it is ready. This 'registration' can be simulated by sending the actual message server object through to the Proxy. It will also mean that the proxy may no longer delete the actual message server. An alternative implementation to using a pointer to the actual message server would be to:

- make use of a shared pointer; or
- make the actual message server a Singleton.

Figure 2 provides a possible solution to the description given above.

An example of how the proxy can be implemented to achieve the requirements for the example in version 2 is given in the following listing.

```
class ProxyMessageServer : public MessageServer {
    ActualMessageServer* implementation;
    bool connected;
    bool registered;
```

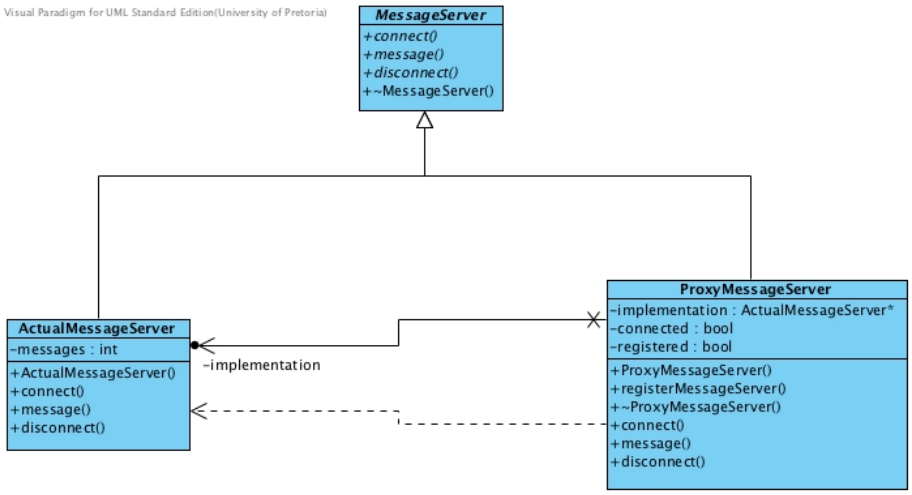

Figure 2: Proxy message server example 2

```
public:

    ProxyMessageServer() {
        registered = false;
    }

    void registerMessageServer(ActualMessageServer* ms) {
        implementation = ms;
        registered = true;
        connected = false;
    }

    ~ProxyMessageServer() {
    }
    void connect() {
        if (registered && !connected) {
            implementation->connect();
            connected = true;
        }
    }
    bool message(string msg) {
        if (registered && connected)
            return implementation->message(msg);
    }

    void disconnect() {
        if (registered && connected) {
            implementation->disconnect();
            connected = false;
        }
    }
```

};

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1995.

[2] Wikipedia. Smart pointer — wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Smart_pointer`, 2013. [Online; accessed 14-October-2013].