FE545 Design, Patterns and Derivatives Pricing

The Factory & Patterns Revisited

Steve Yang

Stevens Institute of Technology

steve.yang@stevens.edu

04/03/2023

Overview

The Problem

- Suppose we wish to design an interface which is a little more sophisticated than those we have used so far. The user will input the name of a pay-off and a strike, and the program will then price a vanilla option with that pay-off. We therefore need a conversion routine from strings and strikes to pay-offs.
- One simple solution is to write a function that takes in the string and the strike, checks against all known types of pay-offs and when it comes across the right one, creates a pay-off of the right type. We would probably implement this via a switch statement, and have to include the header files for all possible forms of pay-off, and every time we added a new pay-off we would have to modify the switch statement. Clearly, this solution would violate the open-closed principle.
- Our solution is a design pattern known as the *factory pattern*. It is so called because it can be used to manufacture objects.

The Solution Idea

- Our solution requires each type of pay-off to tell the factory that it exists, and to give the factory a blueprint for its manufacture.
- The key lies in global variables which are initialized when the program commences before anything else happens. If we define a class in such a way that initializing a global variable of that class registers a pay-off class with the factory, then we have achieved what we wanted.
- For each pay-off class, we write an auxiliary class whose constructor registers the pay-off class with our factory, and we declare a global variable of the auxiliary class.
- We also need a factory object for these auxiliary classes to talk to. In effect, we have a registration function contains a static variable which the factory, then on the first call to the registration function the factory comes into existence.

The Singleton Pattern

- What we need is a factory class for which an object exists as soon as it is required, and for this object to exist until the end of the program.
- We also do not want any other factory objects to exist as they will just confuse matters; everything must be registered with the built by the same factory.
- The singleton pattern gives a way of creating a class with these properties. The first thing is that all constructors and assignment operators are made private. This means the factories can only be created from inside methods of the class this gives us firm control over the existence of factory objects.
- In order to get the one class object that we need, we define very simple method that defines a class object as a static variable. Thus if our class is called *PayOffFactory*, we define a class method *Instance* as follows:

The Singleton Pattern

PayOffFactory.h

```
#include <map>
#include <string>

class PayOffFactory
{
public:
    typedef PayOff*  (*CreatePayOffFunction)(double);

    static PayOffFactory& Instance();
    void RegisterPayOff(std::string, CreatePayOffFunction);
    // allows the possibility of returning a null pointer if the identity strin
    PayOff* CreatePayOff(std::string PayOffID, double Strike);
    ~PayOffFactory(){};

private:
    std::map<std::string, CreatePayOffFunction> TheCreatorFunctions;
    PayOffFactory(){}
    PayOffFactory(const PayOffFactory&){}
    PayOffFactory& operator = (const PayOffFactory&){return *this;}
};
```

The Singleton Pattern

- The first time that *Instance* is called, it creates the *static* data member *theFactory*. As it is a member function, it can do this by using the private default constructor.
- Every subsequent time the *Instance* is called, the address of the already-existing static variable *theFactory* is returned.
- Thus *Instance* creates precisely one *PayOffFactory* object which can be accessed from anywhere by calling *PayOffFactory::Instance()*.
- The meaning of *static* here for a function is quite different from the meaning above for a variable; for a function it means that the function can be called directly without any attachment to an object.
- Note that the name singleton pattern was chosen because precisely on object from the class exists.

Coding the Factory

- How will registration work? Upon Registration, we need to know the string identifier for the specific pay-off class and the pointer to the function which actually creates the object in question.
- There will therefore be the arguments for the registration method. The factory will need to store this information for when the create pay-off method is called.
- There is a container in the standard template library which is designed for associating identifiers to objects. This container is called the map class.
- Finally, we need a method which turns a string plus a strike into a *PayOff* object.

Coding the Factory

PayOffFactory.cpp

```
#include "PayOffFactory.h"
#include <iostream>
using namespace std;

void PayOffFactory::RegisterPayOff(string PayOffId,
                                   CreatePayOffFunction CreatorFunction)
{
    // add one map: sting <-> payoff
    TheCreatorFunctions.insert(pair<string,CreatePayOffFunction>
                               (PayOffId,CreatorFunction));
}
```

Coding the Factory

PayOffFactory.cpp

```
PayOff* PayOffFactory::CreatePayOff(string PayOffId,
                                    double Strike)
{
    map<string, CreatePayOffFunction>::const_iterator
    i = TheCreatorFunctions.find(PayOffId); // serach through the map space to
    if  (i == TheCreatorFunctions.end())
    {
        std::cout << PayOffId
        << " is an unknown payoff" << std::endl;
        return NULL;
    }
    // a pair contains two public data member: first and second
    return (i->second)(Strike); // return an payoff object
}

PayOffFactory& PayOffFactory::Instance()
{
    static PayOffFactory theFactory;
    return theFactory;
}
```

Coding the Factory

- A map is a collection of pairs. Recall that a *pair* is a simple class consisting of two public data members known as first and second.
- The Types of these data members are template parameters. When working with a map, first is the key or identifier used to look up the object we wish to find which is stored in second.
- For us this means that the type of the map is *map¡std::string, CreatePayOffFunction¿* and every pair that we use will be of the same type.
- The *insert* method is used to place pairs of strings and *CreatePayOffFucntions* into the map.
- The map has the property that each key is unique so if you insert two pairs with the same key the second is ignored.
- The retrieval is carried out in *CreatePayOff*: a string is passed in and the find method of map is used. This method returns a *const¡terator* pointing to the pair which as the correct key.

Automatic Registration

PayConstructive.h

```
#if defined(_MSC_VER)
#pragma warning( disable : 4786)
#endif

#include <iostream>
#include "PayOff3.h"
#include "PayOffFactory.h"
#include <string>

template <class T>
class PayOffHelper
{
public:
    PayOffHelper(std::string);
    static PayOff* Create(double);
};

template <class T>
PayOff* PayOffHelper<T>::Create(double Strike)
{
    return new T(Strike);
}
```

Automatic Registration

- The helper class we define here has to do two things. It must define a constructor that carries out the registration of the class defined by the template parameters, and it must define a function which will carry out the creation so we have something to use in the registration process.
- The construction takes in a string as an argument; this string will be needed to identify the class being registered.
- The constructor simply first calls *Instance* to get the address of the factory object, and then calls the *RegisterPayOff* method of the factory to carry out the registration.
- The method *Create* defines the function used to create the pay-off object on demand. Note that it is *static* as it should not be associated to any particular class object.
- The function simply calls the constructor for objects of type T with argument *Strike*.

Automatic Registration

- There is something slightly subtle here in that the specification of the template parameter, T, is making the choice of which object to construct.
- Note that we use *new* as we want the created object to persist after the function is finished. One consequence of this is that the object will have to be properly deleted at some point.
- Note that if we were defining a new class, we would probably put this registration in the source file for the class but as we have already defined the call the put classes.
- The registration file is quite short. We define two global variables, *RegisterCall* and *RegisterPut*. These of type *PayOffHelper¡Call¿* and *PayOffHelper¡Put¿*.
- As global variables, they are initialized at the start of the program, and this initialization carries out the registration as required.

Using the Factory

- Now we have done all the set-up work, how do we use the factory? We give a very simple example in PayFactoryMain.cpp.
  1. The important point here is that the name definitions are carried out in the file *PayOffRegistration.cpp*, and this file is not seen directly by any of the other files including the main routine.
  2. If we want to add another **PayOff**, say the forward, we could do so without modifying any of the existing files.
  3. All we have to do is to add the header and source file for the forward, and in a new file *PayOffForwardRegistration.cpp* add the declaration *PayOffHelper¡PayOffForward¿ RegisterForward("forward");*
- This would not require recompilation of any of the original files. We have therefore achieved our original objective of an open-closed pattern.

Factory Class

PayFactoryMain.cpp

```
int main()
{
    double Strike;
    std::string name;

    cout << "Enter strike\n";
    cin >> Strike;

    cout << "\n pay-off name\n";
    cin >> name;

    PayOff* PayOffPtr =
    PayOffFactory::Instance().CreatePayOff(name,Strike);

    if (PayOffPtr != NULL)
    {
        double Spot;
        cout << "\nspot\n";
        cin >> Spot;

        cout << "\n" << PayOffPtr->operator ()(Spot) << "\n";
        delete PayOffPtr;
    }
```

Breakout Exercise: Add Double Digital Option with the Registration

- Add Double Digital option with the registration. The class cannot handle a double digital as it needs two strikes. Modify PayOffConstructable class to accommodate this change. Work out a solution that will handle options with multiple parameters.
  - Expiry: 1
  - Spot: 50
  - Strike: 50
  - Risk free rate: 0.05
  - Dividend: 0.08
- Please use the given classes and provide a solution.

Design Pattern Revisited

- **Creational Patterns** - a pattern that deals primarily with the creation of new objects.
  - **Virtual copy constructor:** We need a copy of an object, we do not know its type so we cannot use the copy constructor so we ask the object to provide a copy of itself.
  - **The factory:** The purpose of this pattern is to allow use to have an object that spits out objects as and when we need them. This means in particular that responsibility for creating objects of the relevant type lies with a single object.
  - **Singleton:** We used the singleton pattern to implement our factory. The big advantage of the singleton pattern is that there is a single copy of the object which is accessible from everywhere, without introducing global variables and all the difficulties they imply.
  - **Monostate:** We have not examined the monostate pattern nor is it covered in Design Patterns. However, it is useful alternative to the singleton pattern. Rather than only allowing one object from the class to exist, we allow an unlimited number but make them all have the same member variables.

Design Pattern Revisited

- Structural Patterns - A structural pattern is one that deals mainly with how classes are composed to define more intricate structures. They allow us to design code that has extra functionality without having to rewrite existing code.
  - **Adapter:** a class that translates an interface into a form that other classes expect. It is most useful when we wish to fit code into a structure for which it was not originally designed.
  - **Bridge:** is similar to the adapter in that it defines an interface, and acts as an intermediary between a client class and the classes implementing the interface. Thus the implementing class can easily be changed without the client class being aware of the change.
  - **Decorator:** allows us to change the behavior of a class at run-time without changing its interface. We add a wrapper class that processes incoming or outgoing messages and then passes them on.

Design Pattern Revisited

- Behavioral Patterns - Behavioral patterns are used for the implementation of algorithms. They allow us to vary aspects of algorithms interchangeably. They can alos allow us to re-use algorithms in wildly unrelated contexts.
  - **Strategy:** we defer an important part of algorithm to an inputed object. This allows us to easily change how this particular part of the algorithm behaves.
  - **Template:** rather than inputting an aspect of the algorithm, we defer part of the algorithm's implementation to an inherited class. The base class thus provides the structure of how the different parts of the algorithm fit together, but does not specify all the details of the implementation. We adopted this approach when designing our exotics Monte Carlo pricer; there we defined the process for the stock price evolution in an inherited class.
  - **Iterator:**is essentially an abstraction of a pointer. As such it should be possible to dereference it, i.e. look at what it points to, increment and decrement it.

Key Points

- The singleton pattern allows us to create a unique global object from a class and provide a way of accessing it.
- The factory pattern allows us to add extra inherited classes to be accessed from an interface without changing any existing files.
- The factory pattern can be implemented using the singleton pattern.
- The standard template library *map* class is a convenient way to associate objects with string identifiers.
- Placing objects in an unnamed namespace is a way of ensuring that they are not accessed elsewhere.
- We can achieve automatic registration of classes by making their registration a side-effect of the creation of global variables from a helper class.