1. Introduction

In this project, we want to solve a single important financial problem: How do we compute the price of a portfolio of complex financial products in a bank? We want to price a portfolio of options using different pricing methods, such as Monte Carlo simulation and closed-form solutions. Since there should be no restrictions on the options that might be in the portfolio, this makes heavy use of object-oriented programming considering scalability and maintainability. We design our own polymorphic classes. These skills we use in this project include object-oriented features of C++ (Inheritance, Encapsulation, and Polymorphism) and mathematical skills but also include testing, debugging, design, and software architecture. Options are financial instruments that are derivatives based on the value of underlying securities such as stocks.

A goal of this sophisticated financial engineering project is to build a C++ library to price a portfolio containing a variety of different options. Since all our option classes have a price function that takes a Black-Scholes Model as a parameter, it will actually be easy to achieve in the world of object oriented programming that revolves around explicit interfaces and runtime polymorphism through virtual functions and dynamic binding. The other way to do this is to use templates and generic programming, but this is not the way of implementation in this project.

In this project, the header file contains function declarations for all the functions we want to be available outside the .cpp file. To achieve information hiding, we keep the contents of our header files down to a minimum. We also try to use the static keyword on all variables and functions that are not in our header file. By marking a function as static we are saying it can only be used in the current source file. We do not want our end users to know about the implementation details.

Moreover, when we pass objects around, we pass data by reference. Using the **const** keyword to get the compiler's help if we do not want the function to modify the parameters passed in.
If a member function doesn't change the object, we mark it as **const**.

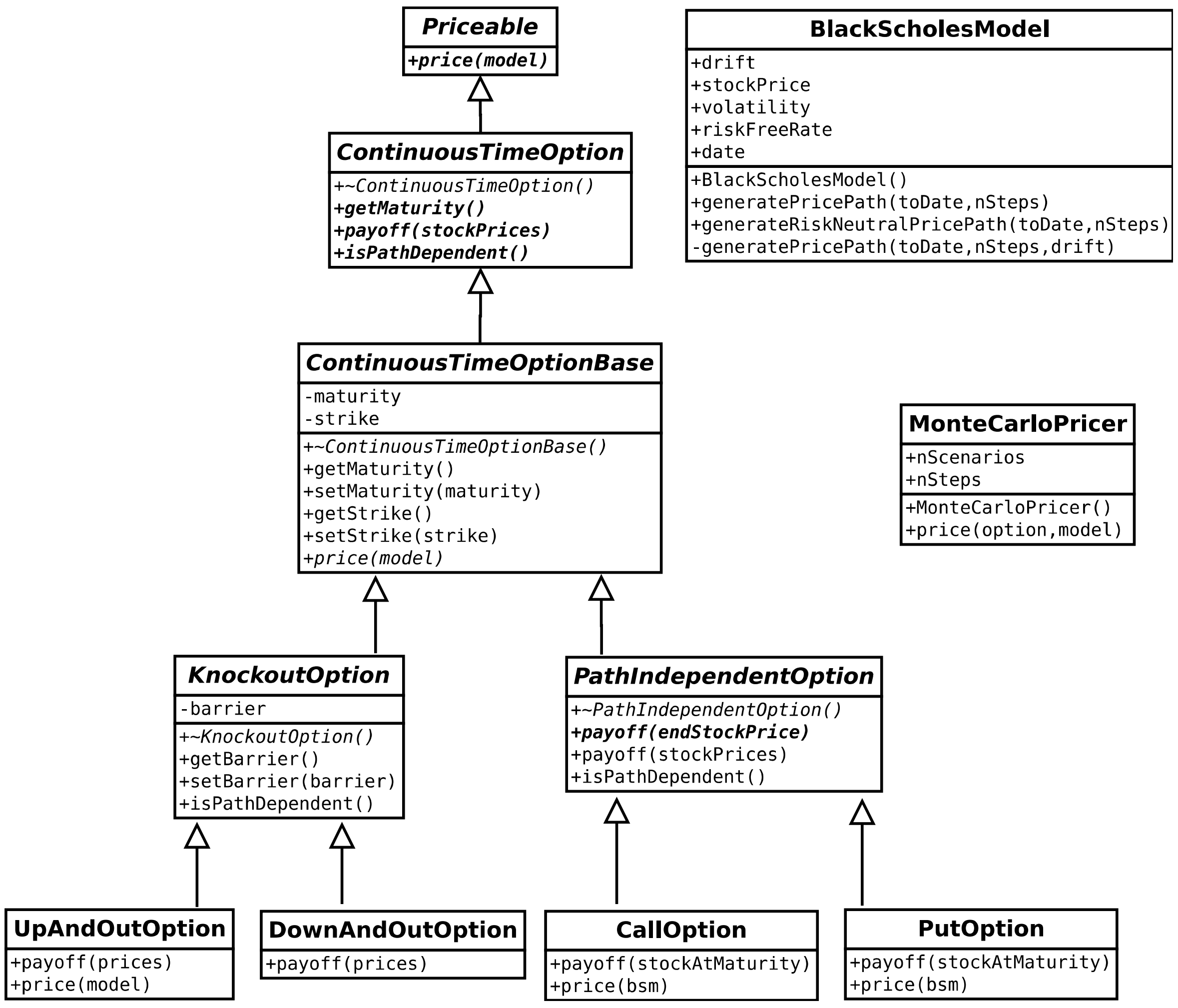
In this project, not putting a class in a header file is our first choice because this gives even more information hiding than making things private. However, if we make the class available to users of our library, we try to make all its data, and as many of its member functions as possible, private.

In the next section, using UML diagram we describe the hierarchy of option classes in our C++ library. In Section 3, we present the design of the portfolio class in a UML diagram while Section 4 is our conclusion. The unit testing framework and math library used for this project is summarized in Appendix A and Appendix B.

2. Hierarchy of Option Classes

Figure 1 shows a UML diagram for the class hierarchy of options in our library. In Figure 1, the arrows mean "extends" or "inherits" or "is a". The boxes denote classes. We follow the standard practice in UML of using italics to indicate abstract classes and roman type to indicate concrete classes. Moreover, we use bold-italic to indicate abstract functions(pure virtual functions), italic to indicate virtual functions(polymorphic), and roman type to indicate final concrete functions.

FIGURE 1: The hierarchy of option classes.



2.1 Interface Priceable and BlackScholesModel

Financially, a portfolio is just a collection of securities such as stocks, bonds, and options in various quantities. We need to decide what class we will use to represent each security. This could be a stock, a bond, a derivative and so forth. The only common feature of these different securities is that they can be priced. This motivates introducing an interface **Priceable** having an abstract function price() that takes a **BlackScholesModel** as a parameter in Figure 1.

BlackScholesModel represents the market data in our financial system. This class only contains the variables associated with the current market data and not variables associated with the options contract. We define a separate class for the option contract which contains a strike and a maturity but does not contain any details about the current market data. The idea behind this design is that we can use the same **BlackScholesModel** to price either a **CallOption** or a **PutOption**. We also add member functions to **BlackScholesModel** to simulate stock prices used by the class **MonteCarloPricer**.

2.2 Interface ContinuousTimeOption and Abstract Class ContinuousTimeOptionBase

By making interface **ContinuousTimeOption** extend **Priceable**, we can ensure that all our options classes implement **Priceable**. A continuous time option has a maturity and we can compute the payoff by looking at the price path up to maturity. This is the idea captured by the

ContinuousTimeOption interface. An interface class typically has no data members, no constructors, a virtual destructor, and a set of pure virtual functions (abstract functions) that specify the interface. Therefore, There are three abstract functions to implement in **ContinuousTimeOption**. Most of the implementation is the same for a **PutOption**, a **CallOption**, etc. Inheritance gives a way of implementing methods for a number of classes at once.

To use inheritance, we begins by defining a so-called "base class" that defines functions we wish to reuse. We call our base class **ContinuousTimeOptionBase**. We use the inlining technique here to implement appropriate get and set methods. Notice that our base class has a virtual destructor. Just like interface classes, any base class must have a virtual destructor.

We also add a new virtual function, price(), to **ContinuousTimeOptionBase** to price an option given the pricing model. To implement the method using Monte Carlo simulation we simply use the following code:

```
double ContinuousTimeOptionBase :: price (const BlackScholesModel& model) const
{
        MonteCarloPricer  pricer ;
        return  pricer.price (*this , model);
}
```

This default implementation works reasonably for all option types, but it isn't necessarily optimal. For example, a **PutOption** is best priced analytically rather than by Monte Carlo simulation. Therefore we would like to be able to override the default implementation. The keyword virtual means that a function may be overridden in a subclass. In the bottom of Figure 1, we can see that both **CallOption** and **PutOption** have a different price function than the default one. When we call price() on an option, the appropriate price method will be automatically selected based on the type of the option. This polymorphism will allow us to write a class that prices an entire Portfolio of options.

2.3 Abstract Class PathIndependentOption and KnockoutOption

In Figure 1, **PathIndependentOption**, an abstract class, contains the code common to **CallOption** and **PutOption** but that is not used by **KnockoutOption**. It provides an inlining implementation of isPathDependent() which always returns false, and has an abstract function, payoff(endStockPrice), to compute the payoff given only the final stock price. Therefore, we can use payoff(stockPrices) to call payoff(endStockPrice) to compute the payoff of a path-independent option.

KnockoutOption, an abstract class, is a type of path dependent options. It has a new private attribute, barrier with get and set inline methods, and also provides an inlining implementation of isPathDependent() which always returns true.

2.4 Concrete Class UpAndOutOption, DownAndOutOption, CallOption, and PutOption

In the bottom of Figure 1, **UpAndOutOption** and **DownAndOutOption** are two types of **KnockoutOption**, and provide overrides for payoff(). **CallOption,** and **PutOption** provide overrides for price(). This is because we know that it would be better to use the Black-Scholes formula to price Call and Put Options than to use the default Monte Carlo implementation.
To sum up, abstract methods and overriding methods are the heart of object-oriented programming, which allow us to write "pluggable" systems. The library users can define new types of options we have never seen before using our pricing library.

3. Design of the Portfolio Class

FIGURE 2: Design of the Portfolio class.

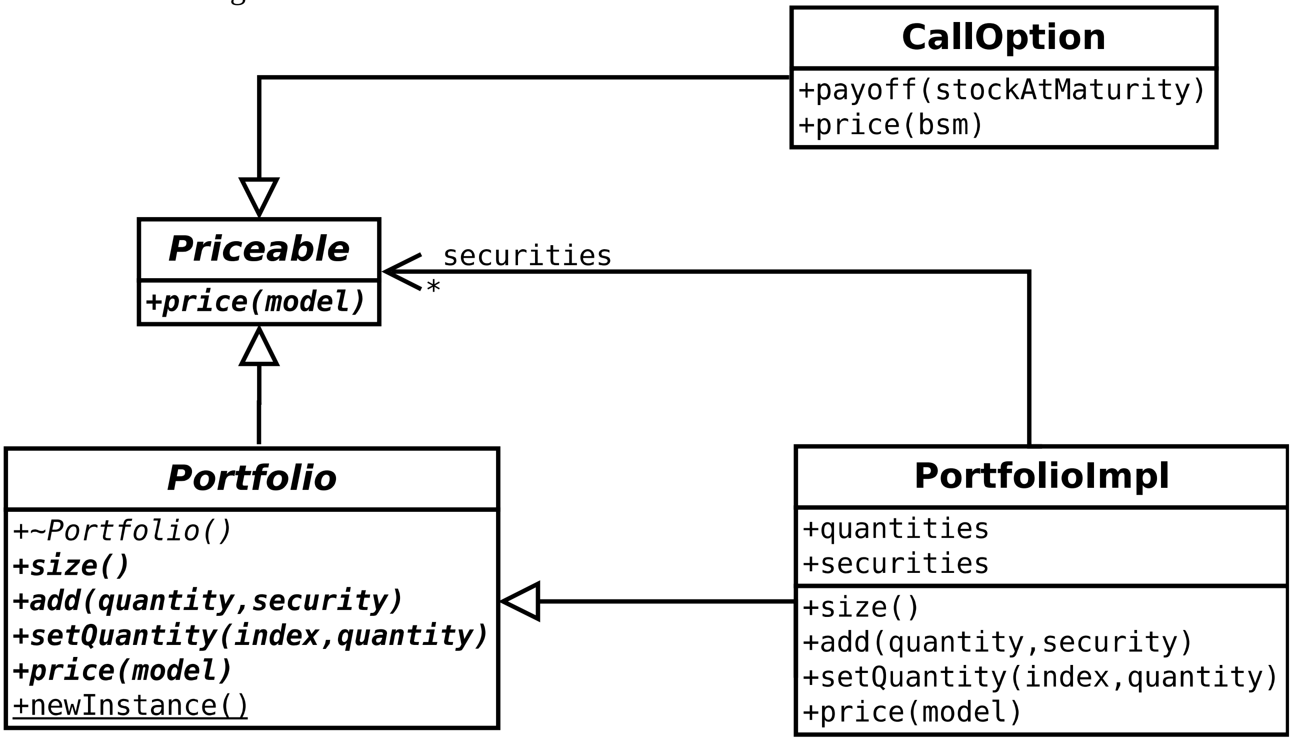


Figure 2 summarizes the classes we have introduced in our C++ library as a UML diagram. The diagram indicates that **PortfolioImpl** implements the interface - **Portfolio** which in turn extends the interface **Priceable**. **Priceable** is also implemented by the class **CallOption**. We have deliberately omitted the full hierarchy of options from our diagram to keep it readable. We have only shown **CallOption** in the diagram above. The full hierarchy of options has been shown in Figure 1.

PortfolioImpl contains a vector of **Priceable** objects; this is indicated by the arrow labeled **securities**. This association relationship is a “has a” relationship. The * symbol on the arrow indicates that a **PortfolioImpl** can have any number of associated securities. In UML we can label the multiplicity of any relationship with a number or range of numbers. The **newInstance** function, a factory method, is underlined to indicate that it is a static function.

3.1 Interface Portfolio

Portfolio class has the following key functions:

(1) a function add() to add a **Priceable** instance together with an associated quantity;
(2) a function setQuantity() to change the quantity held of a given security;
(3) a function price() to compute the value of the Portfolio. This suggests that a **Portfolio** should itself implement the interface **Priceable**. This means that it will be possible to create a **Portfolio** that itself contains Portfolios.

It is not possible for the **Portfolio** implementation to have member variables of type **Priceable** because **Priceable** is an abstract class. For the same reason it is impossible to have a vector of **Priceable** objects. The solution for this is for the **Portfolio** to store data using **shared_ptr**.
This leads us to the following decisions about the design for the Portfolio class.

(1) Our **Portfolio** implementation will hold a vector of **shared_ptr** objects that point to **Priceable** instances.
(2) It also has a vector of quantities.
(3) Since we need to store **shared_ptr** objects, the method to add securities will take a **shared_ptr** to a security instead of a reference to the security.

The users of the **Portfolio** class don't need to know anything about how we will choose to store the data for our **Portfolio**. Therefore it would be best to keep all this inessential detail out of the header file.

3.2 Concrete Class PortfolioImpl

The **Portfolio** class is an abstract class. To create a **Portfolio** a user of this class must call the factory method **newInstance**. This function is guaranteed to return some kind of implementation of the **Portfolio** interface, but the user does not know precisely what class will be returned. Since we don't want the user to even know the member variables of the **Portfolio** objects we return, our factory method has to return a pointer to a **Portfolio**. As usual we use **shared_ptr** to simplify memory management.

The advantage of this is that all implementation details are hidden in the C++ file. Information about private fields and methods are not given to the user. This means we can change all of these implementation details without the user even having to recompile, they just need to link to the latest library.

In header files, the fields of all classes will normally be declared as private to increase encapsulation. Placing implementation class **PortfolioImpl** in the cpp file achieves perfect information hiding without the need for the private keyword. Our implementation class **PortfolioImpl** has a member variable with the rather complex type

```
        vector< shared_ptr<Priceable> > securities;
```

This type is a vector of pointers to **Priceable** objects. It is precisely the data structure we need to store the details about the different securities that our portfolio contains.

This technique of increasing encapsulation by using a factory method, **newInstance**, to provide implementations is called the factory design pattern. One says that this decouples the interface and the implementation. Dependencies between different classes in our program is called coupling. Reducing coupling results in more maintainable code. The following code implements the factory method **newInstance**. This simply needs to return a new **PortfolioImpl** instance.

```
shared_ptr<Portfolio> Portfolio::newInstance()
{
        shared_ptr<Portfolio> ret = make_shared<PortfolioImpl>();
        return ret;
}
```

PortfolioImpl with two member variables implements the interface – **Portfolio**, and all of the implementation code of member functions is very simple and straightforward.

4. Conclusion

Now our **Portfolio** implementation can store any kind of stock options and price it. It will always choose the most efficient method of pricing depending upon the type of options it contains. The end result is an extremely versatile and useful class.

What is more, our **Portfolio** will work equally well with securities we haven't even implemented yet! Using our library we can compute the price of a portfolio of complex financial products in a bank with good scalability and maintainability. The user of our library can provide their own implementations of our classes if they decide to sell novel financial products. Being able to cope with rapidly changing business requirements without having to rewrite all our software is an essential feature of commercial software. It is this "pluggability" that makes object-oriented software so important in the industry.

Appendix A. Unit Testing

In this project, some basic ideas of unit testing are followed.

- Every function should have at least one test.
- All tests should be fully automated
- We assume that code that is not tested does not work.
- We keep our tests forever, and never throw any tests away.
- Write all tests as functions.
- Run all our tests every time we compile our code.

Since C++ has no built-in support for unit testing, in this project we use a very simple testing framework, **testing.h**, which contains various C++ macros. Macros give us access to special variables like the name of the current file and the current line number. These features are useful when writing a testing framework.

To use the testing framework, we include **"testing.h"** in all our .cpp files. For each test that we want to perform, we write a function whose name begins "test". In each .cpp file we also write a single function which calls all other test functions in turn. We name this function after the .cpp file, define it in the .cpp file, and we declare it in the corresponding header file. Finally, in our main method, we call all the test functions defined in the header files. Moreover, to speed up performance of a real system, we can skip all ASSERT, ASSERT_APPROX_EQUAL checks, and disable DEBUG_PRINT when running the release build.

testing.h contains various useful macros and the function declarations, such as ASSERT, ASSERT_APPROX_EQUAL, and DEBUG_PRINT.
testing.cpp contains the definitions of some useful functions declared in **testing.h** which can enable and disable DEBUG_PRINT. To speed up performance of a real system, we can skip all ASSERT, ASSERT_APPROX_EQUAL checks, and disable DEBUG_PRINT when running the release build.

A crucial part of software development is testing that our code works. In the code below, we check the put-call parity formula. The unit test below confirms that this holds for our **Portfolio** class, and also shows how to use our **Portfolio** class. The **Portfolio** class has a very simple price method and every line of that method is already tested by our test of put call parity.

```
static void testPutCallParity() {

   shared_ptr<Portfolio> portfolio = Portfolio::newInstance(); // create a portfolio

   shared_ptr<CallOption> c = make_shared<CallOption>();  // create a call option
   c->setStrike(110);
   c->setMaturity(1.0);

   shared_ptr<PutOption> p = make_shared<PutOption>(); // create a put option
   p->setStrike(110);
   p->setMaturity(1.0);

   portfolio->add( 100, c );   // add call and put options to our portfolio
   portfolio->add( -100, p );

   BlackScholesModel bsm;  // create bsm market data
   bsm.volatility = 0.1;
   bsm.stockPrice = 100;
   bsm.riskFreeRate = 0;

   double expected = bsm.stockPrice - c->getStrike();  // compute the expected result
   double portfolioPrice = portfolio->price( bsm );      // call price method of portfolio, and
                                                         // pass bsm market data to price()

   ASSERT_APPROX_EQUAL( 100*expected, portfolioPrice, 0.0001 );  // call unit test function
                                                                 // to confirm the result
}
```

Appendix B. Math Library

matlib.cpp contains the definitions of various useful financial maths functions, such as normcdf (to compute the cumulative density of the normal distribution) and norminv (to compute its inverse), randn(random numbers generator), and testMatlib(unit testing function for all functions). It also contains definitions of global variables missing from the header.

matlib.h contains function declarations for all the functions we want to be available outside that **matlib.cpp** file. It also contains declarations of global variables, and definitions of constant global variables, such as PI.