The Portfolio Class

The Portfolio Class

- We will achieve the milestone of pricing a Portfolio of derivatives of many different forms.
- This will be easy because of polymorphism.
- No new C++ language features.
- The factory design pattern.

The Priceable interface

```
class Priceable {
public:
    /*  Compute the price of the security in the
        Black--Scholes world */
    virtual double price(
        const BlackScholesModel& model ) const = 0;
};
```

Make `ContinuousTimeOption` extend `Priceable`.

The Portfolio interface

A `Portfolio` class must have the following key functions:

(i) a function to add a `Priceable` instance together with an associated quantity;

(ii) a function to change the quantity held of a given security;

(iii) a function `price` to compute the value of the `Portfolio`.

A `Portfolio` should itself implement the interface `Priceable`

The Portfolio implementation

(i) Our `Portfolio` implementation will hold a vector of `shared_ptr` objects that point to `Priceable` instances.

(ii) It will also have a vector of quantities.

(iii) Because 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.

```
class Portfolio : public Priceable {
public:
    /*  Virtual destructor */
    virtual ~Portfolio() {};
    /* Returns the number of items in the portfolio*/
    virtual int size() const = 0;
    /*  Add a new security to the portfolio,
        returns the index at which it was added */
    virtual int add( double quantity,
        std::shared_ptr<Priceable> security ) = 0;
    /*  Update the quantity at a given index */
    virtual void setQuantity( int index,
                              double quantity ) = 0;
    /*  Compute the current price */
    virtual double price(
        const BlackScholesModel& model ) const = 0;
    /*  Creates a Portfolio */
    static std::shared_ptr<Portfolio> newInstance();
};
```

Factory method

- Portfolio is abstract.
- To obtain an instance you call the *factory method* `newInstance`.
- The user doesn't even know the implementation class, so we return a `shared_ptr`.
- All the implementation details are hidden from the user.
- Decreases *coupling* of code.
  - Makes code compile faster.
  - Makes header files easier for the user to read.

PortfolioImpl

```
class PortfolioImpl : public Portfolio {
public:
    /* Returns the number of items in the portfolio*/
    int size() const;
    /*  Add a new security to the portfolio,
        returns the index at which it was added */
    int add( double quantity,
             shared_ptr<Priceable> security );
    /*  Update the quantity at a given index */
    void setQuantity( int index, double quantity );
    /*  Compute the current price */
    double price(
        const BlackScholesModel& model ) const;

    vector<double> quantities;
    vector< shared_ptr<Priceable> > securities;
};
```

Implement newInstance:

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

Implement add:

```
int PortfolioImpl::add( double quantity,
            shared_ptr<Priceable> security ) {
    quantities.push_back( quantity );
    securities.push_back( security );
    return quantities.size();
}
```

The most interesting method

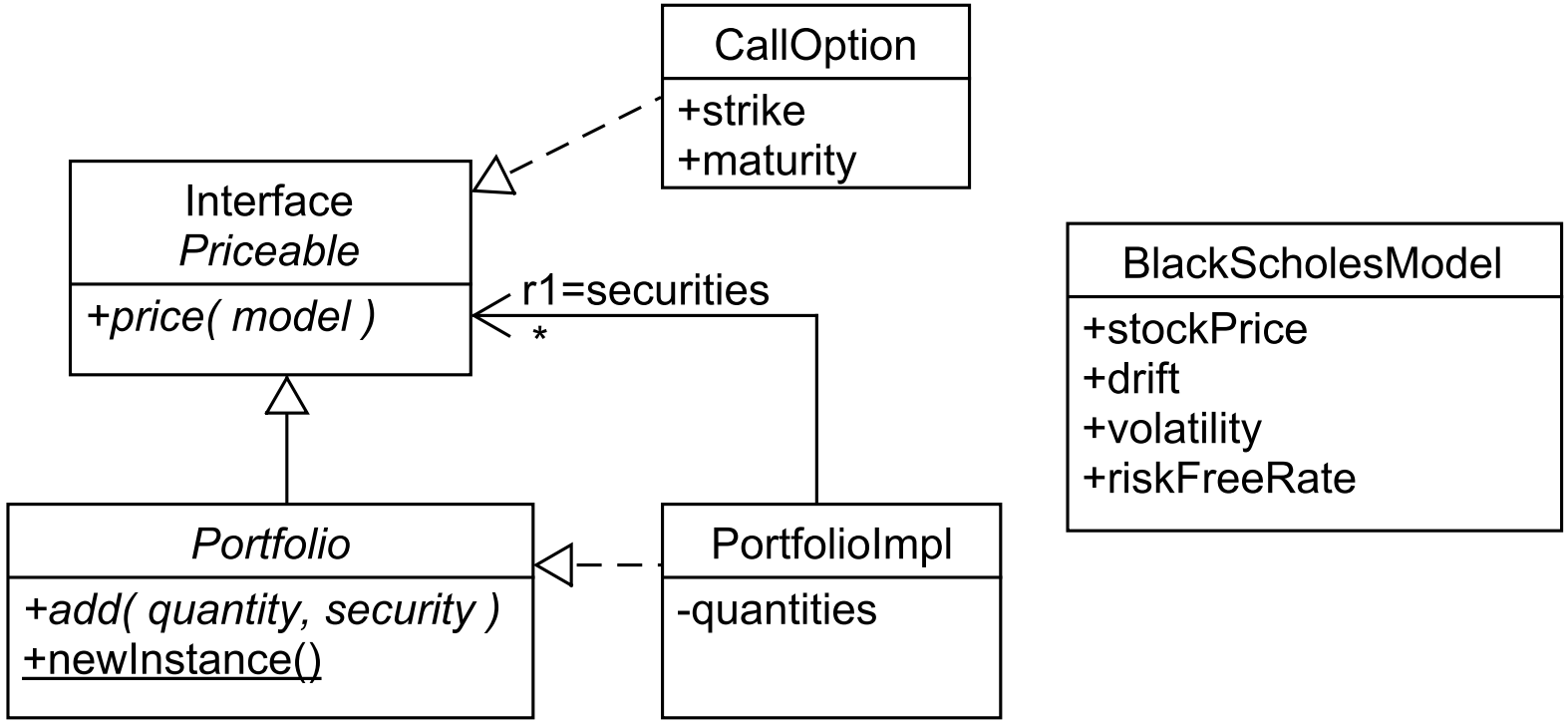
```
double PortfolioImpl::price(
        const BlackScholesModel& model ) const {
    double ret = 0;
    int n = size();
    for (int i=0; i<n; i++) {
        ret += quantities[i]
               * securities[i]->price( model );
    }
    return ret;
}
```

The exciting point is that we can price any `Priceable` object and we will automatically call the best available pricing function.

Test put–call parity

```
static void testPutCallParity() {
    shared_ptr<Portfolio> portfolio
        = Portfolio::newInstance();

    shared_ptr<CallOption> c
        =make_shared<CallOption>();
    c->setStrike(110);
    c->setMaturity(1.0);

    shared_ptr<PutOption> p=make_shared<PutOption>();
    p->setStrike(110);
    p->setMaturity(1.0);

    portfolio->add( 100, c );
    portfolio->add( -100, p );

    BlackScholesModel bsm;
    bsm.volatility = 0.1;
    bsm.stockPrice = 100;
    bsm.riskFreeRate = 0;

    double expected = bsm.stockPrice - c->getStrike();
    double portfolioPrice = portfolio->price( bsm );

    ASSERT_APPROX_EQUAL(100*expected,
        portfolioPrice,0.0001);
}
```

UML


CallOption
+strike
+maturity
Interface
Priceable
+price(model)
r1=securities
*
BlackScholesModel
+stockPrice
+drift
+volatility
+riskFreeRate
Portfolio
+add(quantity, security)
+newInstance()
PortfolioImpl
-quantities

Summary

- Use `shared_ptr` to build sophisticated data structures that store objects long-term.
- Use the *static factory method* design pattern to maximise information hiding and reduce dependencies between files.
- Use object orientation to achieve pluggable code that will not need to be changed even when new requirements come in.