

# Implementing QuantLib

Luigi Ballabio

© 2005, 2006, 2007 Luigi Ballabio.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs License. To view a copy of this license, visit

<http://creativecommons.org/licenses/by-nc-nd/2.0/>

or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Draft



**Attribution-NonCommercial-NonDerivs 2.0**

You are free:

- to copy, distribute, display, and perform the work

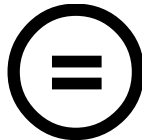
Under the following conditions:



**Attribution.** You must give the original author credit.



**Noncommercial.** You may not use this work for commercial purposes.



**No Derivative Works.** You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

**Your fair use and other rights are in no way affected by the above.**

This is a human-readable summary of the Legal Code (the full license.)

The Legal Code is available at

<http://creativecommons.org/licenses/by-nc-nd/2.0/legalcode>

Draft

## 2. Financial instruments and pricing engines

*Many ways to skin many cats*

THE STATEMENT that a financial library must provide the means to price financial instruments would certainly have appealed to Monsieur de La Palisse. However, that is only a part of the whole problem; a financial library must also provide developers with the means to extend it by adding new pricing functionality.

Foreseeable extension are of two kinds, and the library must allow either one. On the one hand, it must be possible to add new financial instruments; on the other hand, it must be feasible to add new means of pricing an existing instrument. Both kinds have a number of requirements, or in pattern jargon, forces that the solution must reconcile. This chapter details such requirements and describes the design by means of which they are met in QuantLib.

### 2.1. The Instrument class

In our domain, a financial instrument is a concept on its own right. For this reason alone, any self-respecting object-oriented programmer will think to turn it into a base class from which specific instruments will be derived.

It was a natural decision to define such a class in QuantLib. This turned out to have a number of other advantages, the most obvious being that a user is thus able to manage instruments uniformly. This is often done in actual financial practice: for instance, one might be interested in the total value of a portfolio, i.e., a set of instruments. The natural way to obtain the result would be to cycle over the instruments and collect their values; in C++, this maps immediately to storing the instruments in a container of pointers to the base class and inspecting them in turn through their common interface.

### 2.1.1. Interface and requirements

The first design decision under our belt, we turned to the specification of the newly christened `Instrument` class. The first issue was that of finding its public interface, which had to include the methods common to all financial instruments. However, the broad variety of traded assets—which range from the simplest to the most exotic—implies that any method specific to a given class of instruments (say, equity options) is bound not to make sense for some other kind (say, interest-rate swaps.) Therefore, a very few methods were singled out as generic enough to belong to the `Instrument` interface, namely, those returning its present value (possibly with an associated error estimate) and checking whether or not the instrument has expired. The resulting interface is shown in listing 2.1.

As is good practice, the methods were declared as pure virtual ones; but—as Sportin’ Life points out in Gershwin’s *Porgy and Bess*—it ain’t necessarily so. There might be some behavior that can be coded in the base class. In order to find out whether this was the case, we had to analyze what to expect from a generic financial instrument and check whether it could be implemented in a generic way. Two such requirements were found at different times, and their implementation changed during the development of the library; we present them here in their current form.

One is that a given financial instrument might be priced in different ways (e.g., with one or more analytic formulas or numerical methods) without having to resort to inheritance. At this point, the pattern-savvy reader will be thinking “Strategy pattern.” It is indeed so; we devote section 2.2 to its implementation.

The second requirement came from the observation that the value of a financial instrument depends on market data. Such data are by their nature

Listing 2.1: Preliminary interface of the `Instrument` class

---

```
class Instrument {
public:
    virtual ~Instrument();
    virtual Real NPV() const = 0;
    virtual Real errorEstimate() const = 0;
    virtual bool isExpired() const = 0;
};
```

---

## 2.1. *The Instrument class*

9

variable in time, so that the value of the instrument varies in turn; another cause of variability is that a datum can be provided by different sources. We wanted financial instruments to maintain links to such data so that, upon different calls, their methods would access the latest data values and recalculate the results accordingly.

However, with this comes a potential loss of efficiency. For instance, we could monitor the value of a portfolio in time by storing its instruments in a container, periodically poll their values, and add the results. In the outlined implementation, this would trigger recalculation even for those instruments whose inputs did not change. Therefore, we decided to add to the instrument methods a caching mechanism: one that would cause previous results to be stored and only recalculated when any of the inputs change.

### 2.1.2. **Implementation**

The code managing the caching and recalculation of the instrument value was written for a generic financial instrument by means of two design patterns.

When any of the inputs change, the instrument is notified by means of the Observer pattern [?]. The pattern itself is described shortly<sup>1</sup> in appendix A; we describe here the participants.

Obviously enough, the instrument plays the role of the observer while the input data play that of the observables. In order to have access to the new values after a change is notified, the observer needs to maintain a reference to the object representing the input. This might suggest some kind of smart pointer; however, the behavior of a pointer is not sufficient to fully describe our problem. As we already mentioned, a change might come not only from the fact that values from a data feed vary in time; we might also want to switch to a different data feed. Storing a (smart) pointer would give us access to the current value of the object pointed; but our copy of the pointer, being private to the observer, could not be made to point to a different object. Therefore—as explained in the aside on page 11—what we need is the smart equivalent of a pointer to pointer.

Such a feature was implemented in QuantLib as a class template and given the name of `Handle`. Not surprisingly, it is implemented as a smart pointer to a smart pointer. Again, details are given in appendix A; relevant to this discussion is the fact that copies of a given `Handle` share a link to an object. When the link is made to point to another object, all copies are notified and allow their holders to access the new pointee. Furthermore, `Handles` forward any notifications from the pointed object to their observers.

---

<sup>1</sup>This does not excuse the reader from reading the Gang of Four book.

Listing 2.2: Outline of the LazyObject class

---

```

class LazyObject : public Observer {
protected:
    mutable bool calculated_;
    virtual void performCalculations() const = 0;
public:
    void update() { calculated_ = false; }
    virtual void calculate() const {
        if (!calculated_) {
            calculated_ = true;
            try {
                performCalculations();
            } catch (...) {
                calculated_ = false;
                throw;
            }
        }
    }
};

```

---

Finally, classes were implemented which act as observable data and can be stored into Handles. The most basic is the Quote class, representing a single varying market value. Other inputs for financial instrument valuation can include more complex objects such as yield or volatility term structures<sup>2</sup>.

The second part of the problem was to abstract out the code for storing and recalculating cached results, while still leaving it to derived classes to implement any specific calculations. This was done by means of the Template Method pattern [?]. In earlier versions of QuantLib, the functionality was included in the Instrument class itself; later, it was extracted and coded into another class—somewhat lamely called LazyObject—which is now reused in other parts of the library. An outline of the class is shown in listing 2.2.

The code is simple enough. A boolean data member `calculated_` is defined

---

<sup>2</sup>Most likely, such objects ultimately depend on Quote instances, e.g., a yield term structure might depend on the quoted deposit and swap rates used for bootstrapping.

**Aside: pointer semantics**

Storing a copy of a pointer in a class instance gives the holder access to the present value of the pointee, as in the following code:

```
class Foo {
    int* p;
public:
    Foo(int* p) : p(p) {}
    int value() { return *p; }
};

int i=42;
int *p = &i;
Foo f(p);
cout << f.value(); // will print 42
i++;
cout << f.value(); // will print 43
```

However, the stored pointer (which is a copy of the original one) is not modified when the external one is.

```
int i=42, j=0;
int *p = &i;
Foo f(p);
cout << f.value(); // will print 42
p = &j;
cout << f.value(); // will still print 42
```

As usual, the solution is to add another level of indirection. Modifying Foo so that it stores a pointer to pointer gives the class both possibilities.

```
int i=42, j=0;
int *p = &i;
int **pp = &p;
Foo f(pp);
cout << f.value(); // will print 42
i++;
cout << f.value(); // will print 43
p = &j;
cout << f.value(); // will print 0
```

which keeps track of whether results were calculated and still valid. The `update` method, which implements the `Observer` interface and is called upon notification from observables, sets such boolean to `false` and thus invalidates previous results.

The `calculate` method is implemented by means of the Template Method pattern. As explained in that mandatory reading—the Gang of Four book—the constant part of the algorithm (in this case, the management of the cached results) is implemented in the base class; the varying parts (here, the actual calculations) are delegated to a virtual method, namely, `performCalculations`, which is called in the body of the base-class method. Therefore, derived classes will only implement their specific calculations without having to care about caching: the relevant code will be injected by the base class.

The logic of the caching is simple. If the current results are no longer valid, we let the derived class perform the needed calculations and flag the new results as up to date. If the current results are valid, we do nothing.

However, the implementation is not as simple. The reader would be justified in wondering why we had to insert a `try` block and a handler, given that we could have written the body of the algorithm more simply—for instance, as in the following implementation:

```
if (!calculated_) {
    performCalculations();
    calculated_ = true;
}
```

The reason is that there are cases (e.g., when the lazy object is a yield term structure which is bootstrapped lazily) in which `performCalculations` happens to recursively call `calculate`. If `calculated_` were not set to `true`, the `if` condition would still hold and `performCalculations` would be called again, leading to infinite recursion. Setting such flag to `true` prevents this from happening; however, care must now be taken to restore it to `false` if an exception is thrown. The exception is then rethrown so that it can be caught by the installed error handlers.

A few more methods are provided in `LazyObject` which enable users to prevent or force a recalculation of the results. They are not discussed here. We repeat to the interested reader the advice often given by master Obi-Wan Kenobi: “Read the source, Luke.”

The `Instrument` class inherits from `LazyObject`. In order to implement the interface outlined in listing 2.1, it decorates the `calculate` method with code

Listing 2.3: Excerpt of the Instrument class

---

```
class Instrument : public LazyObject {
protected:
    mutable Real NPV_;
public:
    Real NPV() const {
        calculate();
        return NPV_;
    }
    void calculate() const {
        if (isExpired()) {
            setupExpired();
            calculated_ = true;
        } else {
            LazyObject::calculate();
        }
    }
    virtual void setupExpired() const {
        NPV_ = 0.0;
    }
};
```

---

specific to financial instruments. The resulting method is shown in listing 2.3, together with other bits of supporting code.

Once again, the added code follows the Template Method pattern to delegate instrument-specific calculations to derived classes. The class defines an `NPV_` data member to store the result of the calculation; derived classes can declare other data members to store specific results<sup>3</sup>. The body of the `calculate` method calls the virtual `isExpired` method to check whether the instrument is an expired one. If this is the case, it calls another virtual method, namely, `setupExpired`, which has the responsibility of giving meaningful values to the results; its default implementation sets `NPV_` to 0 and can be called by derived classes. The `calculated_` flag is then set to `true`. If the instrument is not expired, the `calculate` method of `LazyObject` is called instead, which in turn

---

<sup>3</sup>The `Instrument` class also defines an `errorEstimate_` member, which is omitted here for clarity of exposition. The discussion of `NPV_` applies to both.

will call `performCalculations` as needed. This imposes a contract on the latter method, namely, its implementations in derived classes are required to set `NPV_` (as well as any other instrument-specific data member) to the result of the calculations. Finally, the `NPV` method ensures that `calculate` is called before returning the answer.

It might be of interest to the reader to explain why `NPV_` is declared as mutable, as this is an issue which often arises when implementing caches or lazy calculations. The crux of the matter is that the `NPV` method is logically a `const` one: calculating the value of an instrument does not modify it. Therefore, a user is entitled to expect that such a method can be called on a `const` instance. In turn, the constness of `NPV` forces us to declare `calculate` and `performCalculations` as `const`, too. However, our choice of calculating results lazily and storing them for later use makes it necessary to assign to one or more data members in the body of such methods. The tension is solved by declaring cached variables as mutable; this allows us (and the developers of derived classes) to fulfill both requirements, namely, the constness of the `NPV` method and the lazy assignment to data members.

### 2.1.3. Example: interest-rate swap

We end this section by showing how a specific financial instrument can be implemented based on the described facilities.

The chosen instrument is the interest-rate swap. As the user surely knows, it is a contract which consists in exchanging periodic cash flows. The net present value of the instrument is calculated by adding or subtracting the discounted cash-flow amounts depending on whether the cash flows are paid or received.

Not surprisingly, the swap is implemented<sup>4</sup> as a new class deriving from `Instrument`. Its outline is shown in listing 2.4. It contains as data members the objects needed for the calculations—namely, the cash flows on the first and second leg and the yield term structure used to discount their amounts—and two variables used to store additional results. Furthermore, it declares methods implementing the `Instrument` interface and others returning the swap-specific results. The class diagram of `Swap` and the related classes is shown in figure 2.1.

In the remainder of this section, we will review the implementation of the class. The relevant methods are shown in listing 2.5.

The fitting of the class to the `Instrument` framework is done in three steps, the third being optional depending on the derived class. The first step is per-

---

<sup>4</sup>The implementation shown in this section is somewhat outdated. After this chapter was first written, the `Swap` class was generalized to allow for more than two legs. However, the original implementation is still used here since it provides a simpler example.

2.1. The Instrument class

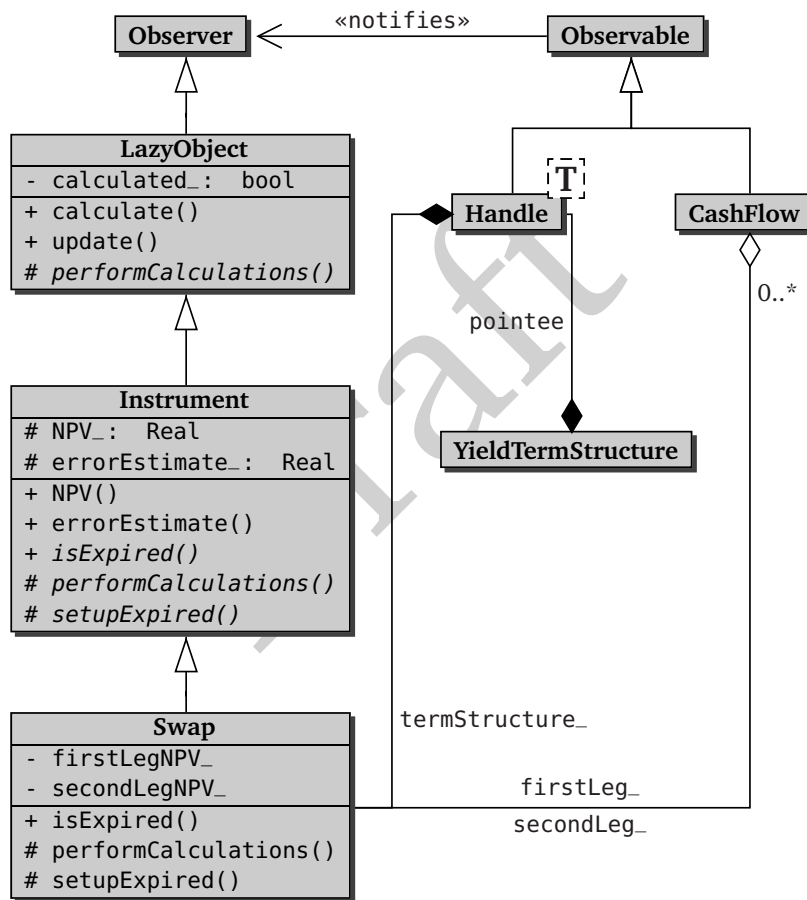


Figure 2.1: Class diagram of the Swap class.

Listing 2.4: Partial interface of the Swap class

---

```

class Swap : public Instrument {
public:
    Swap(const vector<shared_ptr<CashFlow> >& firstLeg,
         const vector<shared_ptr<CashFlow> >& secondLeg,
         const Handle<YieldTermStructure>& termStructure);
    bool isExpired() const;
    Real firstLegBPS() const;
    Real secondLegBPS() const;
protected:
    // methods
    void setupExpired() const;
    void performCalculations() const;
    // data members
    vector<shared_ptr<CashFlow> > firstLeg_, secondLeg_;
    Handle<YieldTermStructure> termStructure_;
    mutable Real firstLegBPS_, secondLegBPS_;
};

```

---

formed in the class constructor. Such method takes as arguments, and copies into the corresponding data members, the two sequences of cash flows to be exchanged and the yield term structure to be used for discounting their amounts. The step itself consists in registering the swap as an observer of both the cash flows and the term structure. As previously explained, this enables them to notify the swap and trigger its recalculation each time a change occurs.

The second step is the implementation of the required interface. The logic of the `isExpired` method is simple enough; its body loops over the stored cash flows checking their payment dates. As soon as it finds a payment which still has not occurred, it reports the swap as not expired. If none is found, the instrument has expired instead.

In the latter case, the `setupExpired` method is called. Its implementation calls the base-class one, thus taking care of the data members inherited from `Instrument`; it then sets to 0 the swap-specific results.

The last required method is `performCalculations`. The calculation is performed by calling two external functions from the `Cashflows` class.<sup>5</sup> The first

<sup>5</sup>If you happen to feel slightly cheated, consider that the point of this example is to show how to

## 2.1. The Instrument class

17

one, namely, `npv`, is a straightforward translation of the algorithm outlined above: it cycles on a sequence of cash flows adding the discounted amount of its future cash flows. We set the `NPV_` variable to the difference of the results from the two legs. The second one, namely, `bps`, calculates the basis-point sensitivity of a sequence of cash flows. We call it once per leg and store the results in the corresponding data members.

The third and final step only needs to be performed if—as in this case—the class defines additional results. It consists in writing corresponding methods (here, `firstLegBPS` and `secondLegBPS`) which ensure that the calculations are (lazily) performed before returning the stored results.

The implementation is now complete. Having been written on top of the `Instrument` class, the `Swap` class will benefit from its code. Thus, it will automatically cache and recalculate results according to notifications from its inputs—even though no related code was written in `Swap` except for the registration calls.

### 2.1.4. Further developments

The reader might have noticed a fallacy in our treatment of the previous example and of the `Instrument` class in general. Albeit generic, the `Swap` class we implemented cannot manage interest-rate swaps in which the two legs are paid in different currencies. A similar problem would arise if the user were to add the values of two instruments whose values are not in the same currency; the user would have to convert manually one of the values to the currency of the other before adding the two.

Such problems stem from a single weakness of the implementation: we used the `Real` type (i.e., a simple floating-point number) to represent the value of an instrument or a cash flow. Therefore, such results miss the currency information which is attached to them in the real world.

The weakness would be removed if we were to express such results by means of the `Money` class. Instances of such class contain currency information; moreover, depending on user settings, they are able to automatically perform conversion to a common currency upon addition or subtraction.

Such development is in our to-do list. However, it is a rather major change, affecting a large part of the code base in a number of ways. This makes us somewhat cautious in forecasting when it will be implemented.

---

package calculations into a class—not to show how to implement such calculations. Your curiosity will be satisfied in a later chapter devoted to cash flows and related functions.

Listing 2.5: Partial implementation of the Swap class

---

```

Swap::Swap(const vector<shared_ptr<CashFlow> >& firstLeg,
           const vector<shared_ptr<CashFlow> >& secondLeg,
           const Handle<YieldTermStructure>& termStructure)
: firstLeg_(firstLeg), secondLeg_(secondLeg),
  termStructure_(termStructure) {
    registerWith(termStructure_);
    vector<shared_ptr<CashFlow> >::iterator i;
    for (i = firstLeg_.begin(); i!= firstLeg_.end(); ++i)
        registerWith(*i);
    for (i = secondLeg_.begin(); i!= secondLeg_.end(); ++i)
        registerWith(*i);
}

bool Swap::isExpired() const {
    Date settlement = termStructure_>referenceDate();
    vector<shared_ptr<CashFlow> >::const_iterator i;
    for (i = firstLeg_.begin(); i!= firstLeg_.end(); ++i)
        if (!(*i)->hasOccurred(settlement))
            return false;
    for (i = secondLeg_.begin(); i!= secondLeg_.end(); ++i)
        if (!(*i)->hasOccurred(settlement))
            return false;
    return true;
}

```

---

Another (and more subtle) fallacy is that the `Instrument` class fails to distinguish explicitly between two components of the abstraction it represents. Namely, there is no clear separation between the data specifying the contract (e.g., the exercise date and strike of an option or the schedule of a bond) and the market data used to price the instrument (e.g., the current value of the option underlying or the current credit spread for the bond issuer.)

The solution might be to collect the first group of data into a term-sheet class. Beside being conceptually clearer, this would provide a useful interface to external functions implementing serialization and deserialization of the instrument—for instance, to and from the FpML format [?].

Listing 2.5 (continued)

---

```

void Swap::setupExpired() const {
    Instrument::setupExpired();
    firstLegBPS_ = secondLegBPS_ = 0.0;
}

void Swap::performCalculations() const {
    NPV_ = - Cashflows::npv(firstLeg_, termStructure_)
          + Cashflows::npv(secondLeg_, termStructure_);
    errorEstimate_ = Null<Real>();

    firstLegBPS_ = - Cashflows::bps(firstLeg_, termStructure_);
    secondLegBPS_ = Cashflows::bps(secondLeg_, termStructure_);
}

Real Swap::firstLegBPS() const {
    calculate();
    return firstLegBPS_;
}

Real Swap::secondLegBPS() const {
    calculate();
    return secondLegBPS_;
}

```

---

## 2.2. Pricing engines

We now turn to the second requirement we stated in the previous section. For any given instrument, it is not always the case that a unique pricing method exists; moreover, one might want to use multiple methods for different reasons. The classic textbook example—the European equity option—will help us make our case. The very same person might want to price it by means of the analytic Black-Scholes formula in order to retrieve implied volatilities from market prices; by means of a stochastic volatility model in order to calibrate the latter and use it for more exotic options; by means of a finite-difference scheme in order to compare the results with the analytic ones and validate one’s finite-difference

implementation; or by means of a Monte Carlo model in order to use the European option as a control variate for a more exotic one.

Therefore, we want it to be possible for a single instrument to be priced in different ways. Of course, it is not desirable to give different implementations of the `performCalculations` method, as this would force one to use different classes for a single instrument type. In our example, we would end up with a base `EuropeanOption` class from which `AnalyticEuropeanOption`, `McEuropeanOption` and others would be derived. This is wrong in at least two ways. On a conceptual level, we would introduce different entities when a single one is needed: a European option is a European option is a European option, as Gertrude Stein said. On a usability level, we would make it impossible to switch pricing methods at run-time.

The solution is to use a strategy pattern, i.e., to let the instrument take an object encapsulating the computation to be performed. We called such an object a *pricing engine*. A given instrument would be able to take any one of a number of available engines (of course corresponding to the instrument type,) pass the chosen engine the needed arguments, have it calculate the value of the instrument and any other desired quantities, and fetch the results. Therefore, the `performCalculations()` method would be implemented roughly as follows:

```
void SomeInstrument::performCalculations() const {
    NPV_ = engine_ -> calculate(arg1, arg2, ... , argN);
}
```

where we assumed that a virtual `calculate` method is defined in the engine interface and implemented in the concrete engines.

Unfortunately, the above approach still leaves something to be desired. Ideally, we would like to implement the dispatching code just once, namely, in the `Instrument` class. However, the code as written is not generic enough, since it relies on the engine accepting a specified number and type of arguments. This is clearly not acceptable, as instrument classes are likely to have data members differing wildly in both number and type. The same goes for the returned results; for instance, an interest-rate swap might return fair values for its fixed rate and floating spread, while the ubiquitous European option might return a number of Greeks.

An interface passing arguments to the engine through a method, as the one outlined above, would thus lead to two undesirable consequences. On the one hand, pricing engines for different instruments would have different interfaces, which would prevent us from defining a single base class. On the other hand, the code for calling the engine would have to be replicated in each instrument

## 2.2. Pricing engines

21

class. This way madness lies.

The solution we chose is one that might cause object-oriented purists to raise an eyebrow; arguments and results are passed and received from the engines by means of opaque structures aptly called `Arguments` and `Results`. Two structures derived from those and augmenting them with instrument-specific data will be stored in any pricing engine; an instrument will write and read such data in order to exchange information with the engine.

Listing 2.6 shows the interface of the resulting `PricingEngine` class, as well as the related `Argument` and `Results` structures and a helper `GenericEngine` template class. The latter implements most of the `PricingEngine` interface, leaving only the implementation of the `calculate` method to developers of specific engines. It can be seen that the `Arguments` and `Results` classes were given methods which ease their use as drop boxes for data. `Arguments::validate` is to be called after input data are written to ensure that their values lie in valid ranges; `Results::reset` is to be called before the engine starts calculating in order to clean previous results.

Armed with our new classes, we can now complete our task of writing a generic `performCalculation` method. Besides the already mentioned Strategy pattern, we will use the Template Method pattern to allow any given instrument to fill the missing bits. The resulting implementation is shown in listing 2.7.

It can be seen that the actual work is split between a number of classes working together towards the goal—the instrument, the pricing engine, and the arguments and results classes. The structure of such task (described in the following paragraphs) might be best understood with the help of the UML sequence diagram shown in figure 2.2.

A call to the NPV method of the instrument eventually triggers (if the instrument is not expired and the relevant quantities need to be calculated) a

### Aside: impure virtual methods

Upon looking at listing 2.7, the reader might wonder why the `setupArguments` method is defined as throwing an exception rather than declared as a pure virtual method. The reason is not to force developers of new instruments to implement a meaningless method, were they to decide that some of their classes should simply override the `performCalculation` method.

Listing 2.6: Interface of PricingEngine and of related classes

---

```

class PricingEngine : public Observable {
public:
    virtual ~PricingEngine() {}
    virtual Arguments* arguments() const = 0;
    virtual const Results* results() const = 0;
    virtual void reset() const = 0;
    virtual void calculate() const = 0;
};

class Arguments {
public:
    virtual ~Arguments() {}
    virtual void validate() const = 0;
};

class Results {
public:
    virtual ~Results() {}
    virtual void reset() = 0;
};

// ArgumentsType must inherit from Arguments;
// ResultType from Results.
template<class ArgumentsType, class ResultsType>
class GenericEngine : public PricingEngine {
public:
    Arguments* arguments() const { return &arguments_; }
    const Results* results() const { return &results_; }
    void reset() const { results_.reset(); }
protected:
    mutable ArgumentsType arguments_;
    mutable ResultsType results_;
};

```

---

Listing 2.7: Excerpt of the Instrument class

---

```

// basic pricing results
class Value : public virtual Results {
public:
    Value() { reset(); }
    void reset() {
        value = errorEstimate = Null<Real>();
    }
    Real value;
    Real errorEstimate;
};

class Instrument : public LazyObject{
protected:
    boost::shared_ptr<PricingEngine> engine_;
public:
    virtual void performCalculations() const {
        QL_REQUIRE(engine_, "null pricing engine");
        engine_>reset();
        setupArguments(engine_>arguments());
        engine_>arguments()>validate();
        engine_>calculate();
        fetchResults(engine_>results());
    }
    virtual void setupArguments(Arguments*) const {
        QL_FAIL("setupArguments() not implemented");
    }
    virtual void fetchResults(const Results* r) const {
        const Value* results = dynamic_cast<const Value*>(r);
        QL_ENSURE(results != 0,
            "no results returned from pricing engine");
        NPV_ = results->value;
        errorEstimate_ = results->errorEstimate;
    }
};

```

---

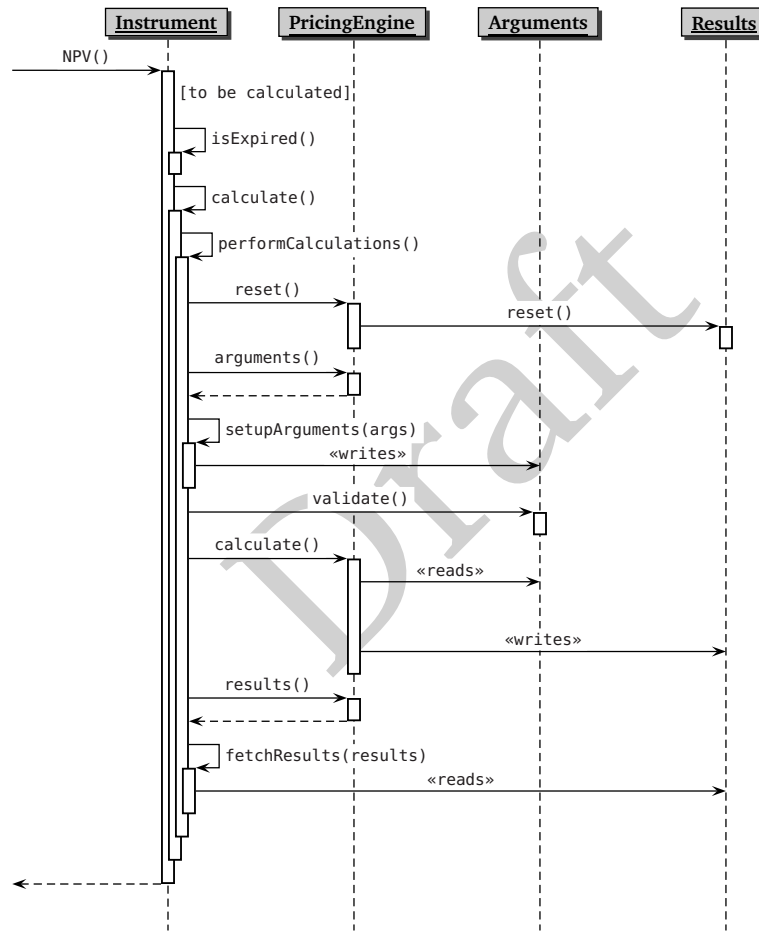


Figure 2.2: Sequence diagram of the interplay between instruments and pricing engines.

call to its `performCalculations` method. Here is where the interplay between instrument and pricing engine begins. First of all, the instrument verifies that an engine is available, aborting the calculation if this is not the case. If one is found, the instrument prompts it to reset itself. The message is forwarded to the instrument-specific result structure by means of its `reset` method; after it executes, the structure is a clean slate ready for writing the new results.

At this point, the Template Method pattern enters the scene. The instrument asks the pricing engine for its argument structure, which is returned as a pointer to `Arguments`. The pointer is then passed to the instrument's `setupArguments` method, which acts as the variable part in the pattern. Depending on the specific instrument, such method verifies that the passed argument is of the correct type<sup>6</sup> and proceeds to fill its data members with the correct values. Finally, the arguments are asked to perform any needed checks on the newly-written values by calling the `validate` method.

The stage is now ready for the Strategy pattern. Its arguments set, the chosen engine is asked to perform its specific calculations, implemented in its `calculate` method. During such processing, the engine will read the inputs it needs from its argument structure and write the corresponding outputs into its results structure.

After the engine has completed its work, the instrument returns in command and the Template Method pattern continues unfolding. The called method, `fetchResults`, must now ask the engine for the results, downcast them to gain access to the contained data—yes, we do see your eyebrow is out of place; we'll come back to this issue later on—and copy such values into its own data members. The `Instrument` class defines a default implementation which fetches the results common to all instruments; derived classes might extend it to read specific results.

### 2.2.1. Example: plain-vanilla option

At this point, an example is necessary to show how the described facilities can be used to implement one's own instruments. However, this purpose is somewhat in conflict with those of a library writer, namely, to find and abstract concepts common to a number of instruments in order to reuse code and make it easier to build new classes upon existing ones. For instance, although a class exists in `QuantLib` which implements plain-vanilla options—i.e., simple call and put equity options with either European, American or Bermudan exercise—such class is actually the lowermost leaf of a class hierarchy. Having the `Instrument` class

---

<sup>6</sup>This is done by means of a `dynamic_cast`. Yes, I know. Just bear with us a little longer.

Listing 2.8: Interface of the VanillaOption class

---

```

class VanillaOption : public OneAssetStrikedOption {
public:
    // accessory classes
    class arguments;
    class results;
    class engine;
    // constructor
    VanillaOption(const boost::shared_ptr<StochasticProcess>&,
                 const boost::shared_ptr<Payoff>&,
                 const boost::shared_ptr<Exercise>&,
                 const boost::shared_ptr<PricingEngine>& e =
                 boost::shared_ptr<PricingEngine>());

    // implementation of instrument method
    bool isExpired() const;
    void setupArguments(Arguments*) const;
    void fetchResults(const Results*) const;
    // accessors for option-specific results
    Real delta() const;
    Real gamma() const;
    Real theta() const;
    // ...more greeks
protected:
    void setupExpired() const;
    // option data
    boost::shared_ptr<StochasticProcess> process_;
    boost::shared_ptr<Payoff> payoff_;
    boost::shared_ptr<Exercise> exercise_;
    // specific results
    mutable Real delta_;
    mutable Real gamma_;
    mutable Real theta_;
    // ...more
};
    
```

---



Listing 2.10: Implementation of the VanillaOption class

---

```

VanillaOption::VanillaOption(
    const boost::shared_ptr<StochasticProcess>& process,
    const boost::shared_ptr<StrikedTypePayoff>& payoff,
    const boost::shared_ptr<Exercise>& exercise,
    const boost::shared_ptr<PricingEngine>& engine)
: payoff_(payoff), exercise_(exercise),
  stochasticProcess_(process) {
    if (engine)
        setPricingEngine(engine);
    registerWith(stochasticProcess_);
}

bool VanillaOption::isExpired() const {
    Date today = Settings::instance().evaluationDate();
    return exercise_>lastDate() < today;
}

void VanillaOption::setupExpired() const {
    Instrument::setupExpired();
    delta_ = gamma_ = theta_ = ... = 0.0;
}

```

---

at its root, such hierarchy specializes it first with an `Option` class, then again with a `OneAssetOption` class generalizing options on a single underlying, passing through another class or two until it finally defines the `VanillaOption` class we are interested in. There are good reasons for this proliferation; for instance, the code in the `OneAssetOption` class can naturally be reused for, say, Asian options, while that in the `Option` class lends itself for reuse when implementing all kinds of basket options. Unfortunately, this causes the code for pricing a plain option to be spread among all the members of the described inheritance chain, which does not make for an extremely clear example. Therefore, we had to come to a compromise. In this section, we will describe a synthesized `VanillaOption` class

Listing 2.10 (continued)

---

```

void VanillaOption::setupArguments(Arguments* args) const {
    VanillaOption::arguments* arguments =
        dynamic_cast<VanillaOption::arguments*>(args);
    QL_REQUIRE(arguments != 0, "wrong argument type");

    arguments->stochasticProcess = stochasticProcess_;
    arguments->exercise = exercise_;
    arguments->payoff = payoff_;

    arguments->stoppingTimes.clear();
    for (Size i=0; i<exercise->dates().size(); i++) {
        arguments->stoppingTimes.push_back(
            stochasticProcess->time(exercise->date(i)));
    }
}

void VanillaOption::fetchResults(const Results* r) const {
    Instrument::fetchResults(r);
    const VanillaOption::results* results =
        dynamic_cast<const VanillaOption::results*>(r);
    QL_ENSURE(results != 0, "wrong result type");
    delta_ = results->delta;
    ... // other Greeks
}

```

---

with the same implementation as the one in the library, but inheriting directly from the `Instrument` class; all code implemented in the intermediate classes will be shown—inlined, that is—as if it were implemented in the example class rather than inherited.

Listing 2.8 shows the synthesized interface of our vanilla-option class. It declares the required methods from the `Instrument` interface, as well as accessors for additional results, namely, the greeks of the options; as pointed out in the previous section, the corresponding data members are declared as mutable so that their values can be set in the logically constant `calculate` method.

Besides its own data and methods, `VanillaOption` declares a number of

Listing 2.11: Sketch of an engine for the VanillaOption class

---

```

class AnalyticEuropeanEngine
  : public VanillaOption::engine {
public:
  void calculate() const {
    QL_REQUIRE(
      arguments_.exercise->type() == Exercise::European,
      "not an European option");
    shared_ptr<BlackScholesProcess> process =
      dynamic_pointer_cast<BlackScholesProcess>(
        arguments_.stochasticProcess);
    QL_REQUIRE(process, "Black-Scholes process needed");
    ... // other requirements

    Real spot = process->stateVariable()->value();
    ... // other needed quantities

    BlackFormula black(forwardPrice, riskFreeDiscount,
      variance, payoff);

    results_.value = black.value();
    results_.delta = black.delta(spot);
    ... // other greeks
  }
};

```

---

accessory classes, namely, the specific argument and result structures and a base pricing engine. Such classes are defined as inner classes to highlight the relationship between them and the option class; their interface is shown in listing 2.9.

Two comments can be made on such accessory classes. The first is that, opposite to what we said in our introduction to the example, we didn't inline all data members into the results class. This was done in order to point out an implementation detail. It might occur to a user to define structures holding a few related and commonly used results; such structures can then be reused by means of inheritance, as exemplified by the Greeks structure that is here composed

## 2.2. Pricing engines

31

with `Value` to obtain the final structure. In this case, virtual inheritance from `Results` should be used to avoid the infamous inheritance diamond.

The second comment is that, as shown, it is sufficient to inherit from the template class `GenericEngine` (instantiated with the right argument and result types) to provide a base class for instrument-specific pricing engines. We will see that derived classes only need to implement their `calculate` method.

We now turn to the implementation of the `VanillaOption` class, shown in listing 2.10. Its constructor takes a few objects defining the instrument. Most of them will be described in later chapters or in appendix A. For the time being, we just mention that the stochastic process contains information related to the dynamics of the underlying (namely, present value, risk-free rate, dividend yield, and volatility;) the payoff contains the strike and type (i.e., call or put) of the option; and the exercise contains information on the exercise date (or dates) and the type of exercise (European, American, or Bermudan.) The passed arguments are stored in the corresponding data members; moreover, the instrument registers itself with each passed observable and sets itself the passed engine.

The methods related to expiration are simple enough; `isExpired` checks whether the latest exercise date is passed, while `setupExpired` calls the base-class implementation and sets the instrument-specific data to 0.

The `setupArguments` and `fetchResults` methods are a bit more interesting. The former starts by downcasting the generic `Argument` pointer to the actual type required, raising an exception if another type was passed; it then turns to the actual work. On the one hand, some of the data members are just copied verbatim into the corresponding argument slots. On the other hand, there might be some calculations that most engines are likely to need. Such calculations can be performed here; in this case, the transformation of exercise dates into the corresponding time.

The `fetchResults` method is the dual of the previous one. It also starts by downcasting the passed `Results` pointer; after verifying its actual type, it just copies the results into his own data members.

Albeit simple, the above implementation is everything we needed to have a working instrument—working, that is, once it is set an engine which will perform the required calculations. Such an engine is sketched in listing 2.11 and implements the analytic Black-Scholes-Merton formula for European options.

Once again, the actual calculations are hidden behind the interface of another class, namely, the `BlackFormula` class. However, the code shown has enough detail for us to show a few relevant features.

The method starts by verifying a few preconditions. This might come as a

**Aside: a mixed approach**

It is possible for an instrument to provide or inherit a default calculation in its `performCalculations` method while giving the user the possibility to change it by setting a different engine. One obvious way to do it is to set a default engine in the instrument constructor. Another way is the one implemented in the `VanillaSwap` instrument, which inherits from `Swap` the calculation described in section 2.1.3. Its `performCalculations` code is implemented as follows:

```
void VanillaSwap::performCalculations() const {
    if (engine_) {
        Instrument::performCalculations();
    } else {
        Swap::performCalculations();
        ... // copy results in data members
    }
}
```

surprise, since the arguments of the calculations are already validated by the time the `calculate` method is called. However, any given engine can have further requirements to be fulfilled before its calculations can be performed. In the case of our engine, two such requirements are that the option is European and that the process followed by its underlying is a Black-Scholes one.

In the middle section of the method, the engine extracts from the passed arguments any information not already presented in digested form. Shown here is the retrieval of the spot price of the underlying; other quantities needed by the engine, e.g., the values of the risk-free rate and the term volatility of the underlying, are also extracted<sup>7</sup>.

Finally, the calculation is performed and the results are stored in the corresponding slots of the results structure. This concludes both the `calculate` method and the example.

**2.2.2. Further developments**

The reader—his eyebrows now lowered by sheer exhaustion—might still be wondering whether the need for downcasting can be eliminated. It could; but it

<sup>7</sup>The interested reader can find the full code of the engine in the QuantLib sources.

## 2.2. Pricing engines

33

might cost more than it is worth.

For any given instrument to be given arguments of the correct type, and for the relevant methods to be written in a generic way, some kind of template programming must be used; moreover, the use of the Template Method pattern requires some kind of polymorphism. This suggests the use of the Curiously Recurring Template pattern; a sketch of the possible implementation is shown in listing 2.12.

The `GenericEngine` class would be modified so that its inspectors return the specific argument and result structures, instead of the opaque base classes. This removes the need for downcasting; however, it also forces the instrument to match such types exactly.

Therefore, the specific argument and result types are passed as template arguments to the `Instrument` class, together with the specific instrument type. Most of the class remains unchanged, except for the `setupArguments` method and its dual `fetchResults` which are no longer declared in the `Instrument` interface. Accordingly, the `performCalculations` body must be modified: the `this` pointer must be statically cast to the specific instrument type so that calls to such methods can be bound. Finally, the developer of an instrument must close the cycle as shown in the listing: the specific instrument must both be inherited from `Instrument` and passed as a template argument to the same class, and the required methods must be defined. Also, the argument and result structures must be declared before the instrument in order to be available at the point of template instantiation.

Albeit working, this implementation seems to us to provide no real advantage, while making it more cumbersome for developers to create new instruments. Furthermore, it makes it impossible to derive from a class once the template cycle is closed; for instance, subclasses of the `Swap` class in the listing would have no way to modify or extend the calculation. This might prevent one, for instance, from coding a bond class usable on its own and a fixed-coupon bond class deriving from the former. Therefore, it is likely that the current `Instrument` implementation will remain unchanged in this respect.

Listing 2.12: Sketch of an Instrument template class

---

```

template <class A, class R>
class GenericEngine {
public:
    A& arguments() const { return arguments_; }
    const R& results() const { return results_; }
private:
    A arguments_;
    R results_;
};

template <class T, class A, class R>
class Instrument {
public:
    typedef A arguments;
    typedef R results;
    typedef GenericEngine<arguments,results> engine;
    void performCalculations() const {
        engine_→reset();
        static_cast<const T*>(this)→setupArguments(
            engine_→arguments());
        engine_→arguments().validate();
        engine_→calculate();
        static_cast<const T*>(this)→fetchResults(
            engine_→results());
    }
private:
    boost::shared_ptr<engine> engine_;
};

class SwapArguments { ... };
class SwapResults { ... };

class Swap
: public Instrument<Swap,SwapArguments,SwapResults> {
public:
    void setupArguments(arguments&) { ... }
    void fetchResults(const results&) { ... }
};

```

---

Draft

Draft