Implementing QuantLib

Luigi Ballabio

Financial instruments and pricing engines

Many ways to skin many cats

$\mathcal{T}$HE 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 extensions 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 that allows QuantLib to satisfy them.

2.1. The `Instrument` class

In our domain, a financial instrument is a concept in its own right. For this reason alone, any self-respecting object-oriented programmer will code it as 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.) Thus, very few methods were singled out as generic enough to belong to the Instrument interface. We limited ourselves to those returning its present value (possibly with an associated error estimate) and indicating 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; I 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; I 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 variable in time, so that the value of the instrument varies in turn; another cause of variability is any single market datum can be provided by different sources. We wanted financial instruments to maintain links to these sources so that, upon different calls, their methods would access the latest values and recalculate the results accordingly; also, we wanted to be able to transparently switch between sources and have the instrument treat this as just another change of the data values.

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

Listing 2.1: Preliminary interface of the Instrument class.

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

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 [13]. The pattern itself is briefly described[1] in appendix A; I 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 I 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, what we need is the smart equivalent of a pointer to pointer. This feature was implemented in QuantLib as a class template and given the name of Handle. 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.

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[2].

Another 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 [13]. In earlier versions of QuantLib, the functionality was included in the Instrument class itself; later, it was extracted and coded into another class—somewhat unsurprisingly called LazyObject—which is now reused in other parts of the library. An outline of the class is shown in listing 2.2.

---

[1]This does not excuse you from reading the Gang of Four book.

[2]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.

8   *Chapter 2. Financial instruments and pricing engines*

Listing 2.2: Outline of the `LazyObject` class.

```
class LazyObject : public virtual Observer,
                   public virtual Observable {
  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;
            }
        }
    }
};
```

The code is simple enough. A boolean data member `calculated_` is defined 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 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. You would be justified in wondering why we had to insert a `try` block setting `calculated_` beforehand and a handler rolling back the change before throwing the exception again. After all,

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;
    }
};
```

we could have written the body of the algorithm more simply—for instance, as in the following, seemingly equivalent implementation, that manages to do without catching and rethrowing exceptions:

```
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. I repeat to the interested reader the advice often given by master Obi-Wan Kenobi: "Read the source, Luke."

Aside: const or not const?

It might be of interest 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.

The `Instrument` class inherits from `LazyObject`. In order to implement the interface outlined in listing 2.1, it decorates the `calculate` method with code 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[3]. 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 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.

2.1.3.   Example: interest-rate swap

I 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

---

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

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_;
};
```

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[4] 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, I 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 performed in the class constructor, which 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.

---

[4]The implementation shown in this section is somewhat outdated. However, the original implementation is still used here since it provides a simple example.
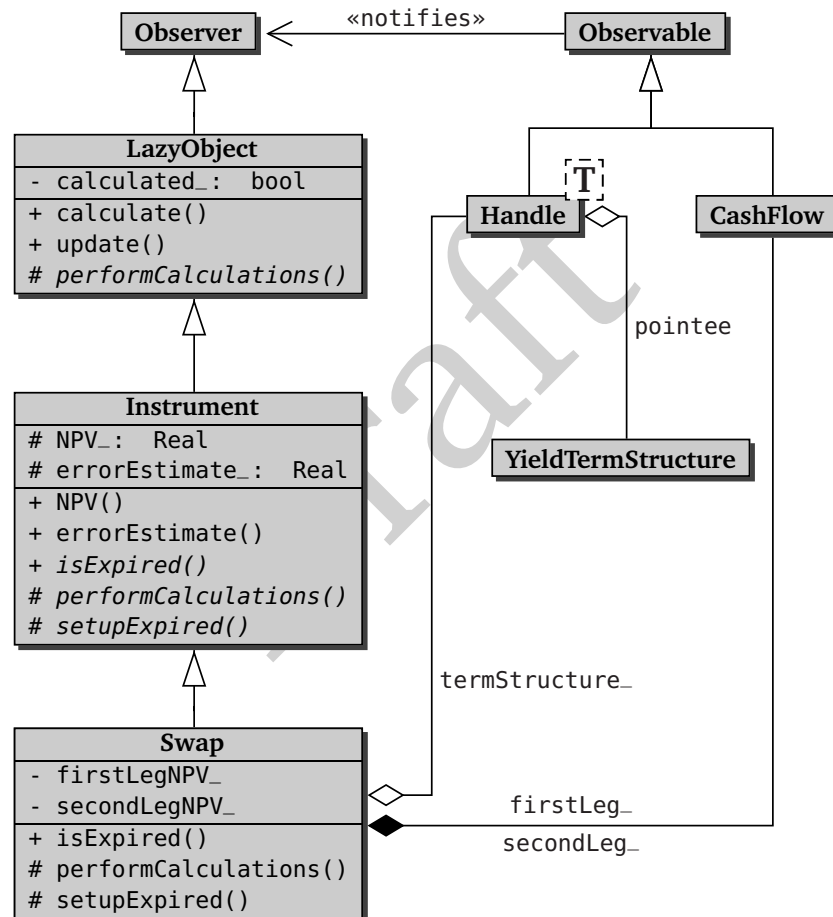
Figure 2.1: Class diagram of the Swap class.

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;
}
```

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. In this case, the setupExpired method will be 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.[5] The first 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.

---

[5]If you happen to feel slightly cheated, consider that the point of this example is to show how to 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 (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_;
}
```

The second one, bps, calculates the basis-point sensitivity (BPS) of a sequence of cash flows. We call it once per leg and store the results in the corresponding data members. Since the result carries no numerical error, the errorEstimate_ variable is set to Null<Real>()—a specific floating-point value which is used as a sentinel value indicating an invalid number.[6]

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.

---

[6]NaN might be a better choice, but the means of detecting it are not portable. Another possibility still to be investigated would be to use boost::optional.

Aside: handles and shared pointers.

You might wonder why the Swap constructor accepts the discount curve as a handle and the cash flows as simple shared pointers. The reason is that we might decide to switch to a different curve (which can be done by invoking the handle's linkTo method) whereas the cash flows are part of the definition of the swap and are thus immutable.

2.1.4.   Further developments

You might have noticed a shortcoming in my 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. Therefore, it won't appear in release 1.0; it might be introduced in the 1.x series in some form, or it might wait for release 2.0 sometime in the future (don't hold your breath for that one, though.—it will take a few years.)

Another (and more subtle) shortcoming is that the Swap 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 (the cash-flow specification) and the market data used to price the instrument (the current discount curve.)

The solution is to store in the instrument only the first group of data (i.e., those that would be in its term sheet) and keep the market data elsewhere.[7] The means to do this are the subject of the next section.

---

[7]Beside being conceptually clearer, this would prove useful to external functions implementing serialization and deserialization of the instrument—for instance, to and from the FpML format [17].

2.2.   Pricing engines

We now turn to the second requirement I 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 me make my 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, it 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, it would make it impossible to switch pricing methods at run-time.

The solution is to use the 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 any 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 class. This way madness lies.

The solution we chose was that arguments and results be 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 its inner `argument` and `results` classes and a helper `GenericEngine` class template. 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 noted that an inner class `Instrument::result` was defined; such class inherits from `PricingEngine::results` and contains the results that have to be provided for any instrument[8]

As for `performCalculation`, it can be seen that the actual work is split between a number of collaborating classes—the instrument, the pricing engine, and the arguments and results classes. The dynamics of such a collaboration (described in the following paragraphs) might be best understood with the help of the UML sequence diagram shown in figure 2.2; the static relationship between the classes (and a possible concrete instrument) is shown in figure 2.3.

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 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

---

[8]The `Instrument::results` class also contains a `std::map` where pricing engines can store additional results. The relevant code is here omitted for clarity.

Listing 2.6: Interface of `PricingEngine` and of related classes.

```cpp
class PricingEngine : public Observable {
  public:
    class arguments;
    class results;
    virtual ~PricingEngine() {}
    virtual arguments* getArguments() const = 0;
    virtual const results* getResults() const = 0;
    virtual void reset() const = 0;
    virtual void calculate() const = 0;
};

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

class PricingEngine::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:
    PricingEngine::arguments* getArguments() const {
        return &arguments_;
    }
    const PricingEngine::results* getResults() const {
        return &results_;
    }
    void reset() const { results_.reset(); }
  protected:
    mutable ArgumentsType arguments_;
    mutable ResultsType results_;
};
```

Listing 2.7: Excerpt of the Instrument class.

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

class Instrument::results
    : public virtual PricingEngine::results {
  public:
   Value() { reset(); }
   void reset() {
       value = errorEstimate = Null<Real>();
   }
   Real value;
   Real errorEstimate;
};
```
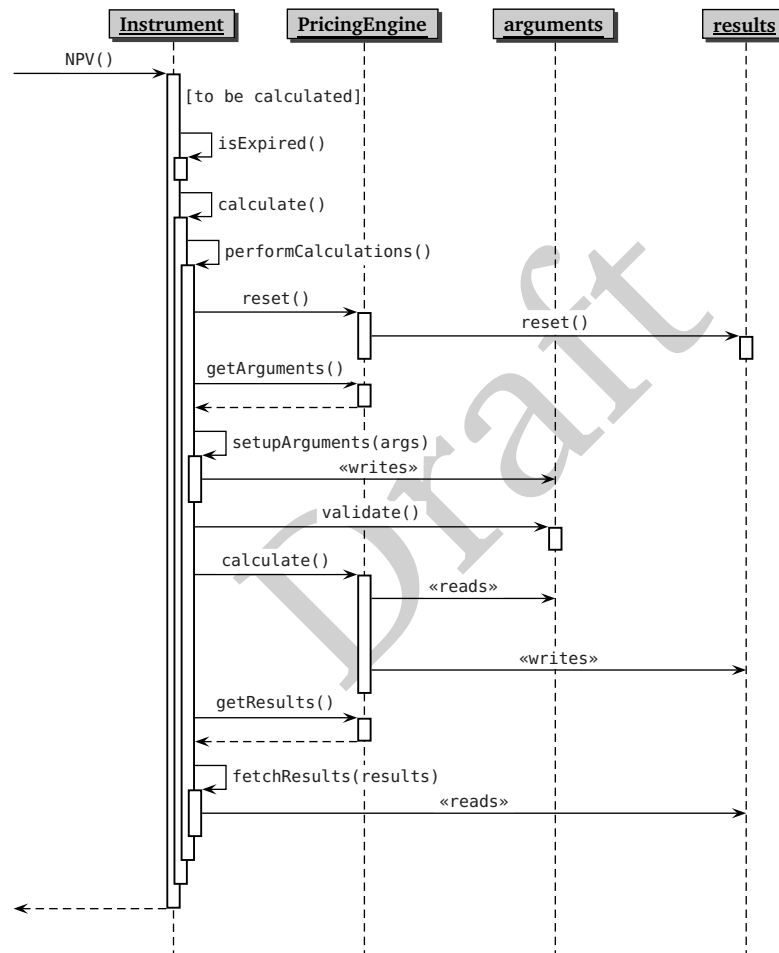
Figure 2.2: Sequence diagram of the interplay between instruments and pricing engines.
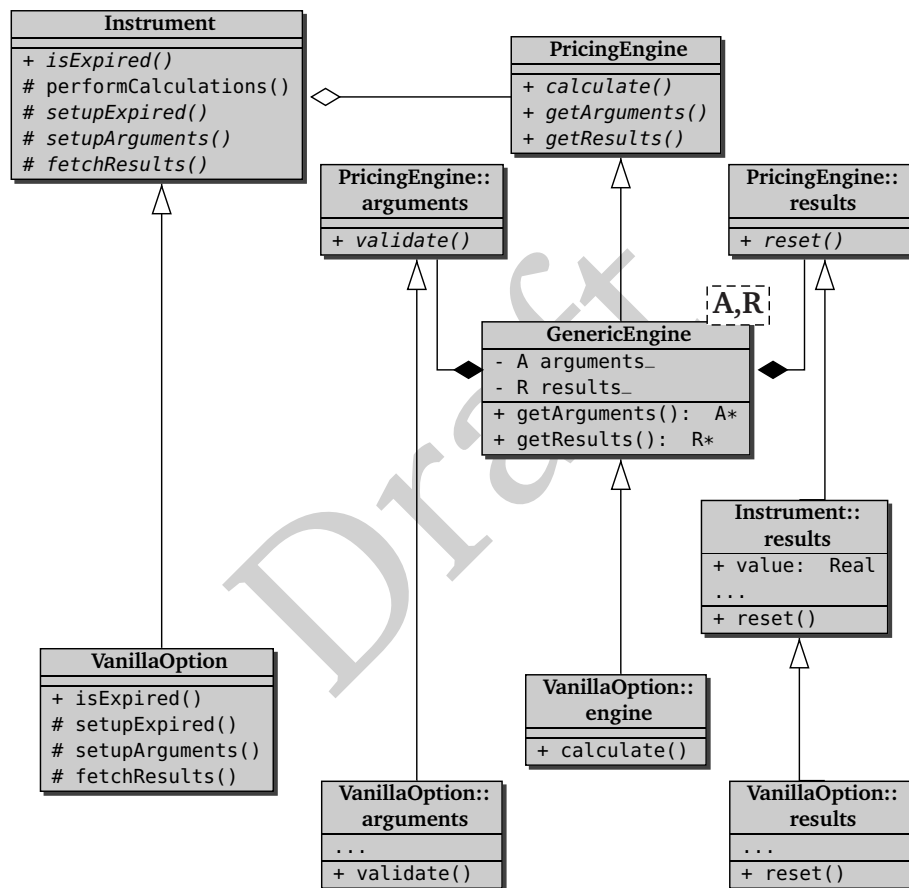
Figure 2.3: Class diagram of `Instrument`, `PricingEngine`, and related classes including a derived instrument class.

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 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 completes its work, the control returns to the `Instrument` instance 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, 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

---

Aside: impure virtual methods.

Upon looking at listing 2.7, you 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 meaning-less method, were they to decide that some of their classes should simply override the `performCalculation` method.

is actually the lowermost leaf of a class hierarchy. Having the `Instrument` class 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, I had to come to a compromise. In this section, I will describe a synthesized `VanillaOption` class 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 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 I said in my introduction to the example, I 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 with `Value` to obtain the final structure. In this case, virtual inheritance from `Results` should be used to avoid the infamous inheritance diamond (see, for instance, section 15.2 of [25].)

The second comment is that, as shown, it is sufficient to inherit from the class template `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, suffice to say that the payoff contains the strike and right (i.e., call or put) of the option, and the exercise contains information on the exercise dates and the variety (i.e., European, American, or Bermudan.) The passed arguments are stored in the

Listing 2.8: Interface of the `VanillaOption` class.

```cpp
class VanillaOption : public Instrument {
  public:
    // accessory classes
    class arguments;
    class results;
    class engine;
    // constructor
    VanillaOption(const boost::shared_ptr<Payoff>&,
                  const boost::shared_ptr<Exercise>&);
    // 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<Payoff> payoff_;
    boost::shared_ptr<Exercise> exercise_;
    // specific results
    mutable Real delta_;
    mutable Real gamma_;
    mutable Real theta_;
    // ...more
};
```

corresponding data members. Also, note that they do not include market data; those will be passed elsewhere.

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. In some cases, the data members are just copied verbatim into the corresponding argument slots. However, it might be the case that the same

Listing 2.9: Interface of the VanillaOption inner classes.

```
class VanillaOption::arguments
    : public PricingEngine::arguments {
  public:
    // constructor
    arguments();
    void validate() const;
    boost::shared_ptr<Payoff> payoff;
    boost::shared_ptr<Exercise> exercise;
};

class Greeks : public virtual PricingEngine::results {
  public:
    Greeks();
    Real delta, gamma;
    Real theta;
    Real vega;
    Real rho, dividendRho;
};

class VanillaOption::results : public Instrument::results,
                                 public Greeks {
  public:
    void reset();
};

class VanillaOption::engine
    : public GenericEngine<VanillaOption::arguments,
                            VanillaOption::results> {};
```

calculations will be needed by a number of engines; setupArguments provides a place to write them just once.

The fetchResults method is the dual of setupArguments. 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. Its constructor takes (and registers itself with) a Black-Scholes stochastic process that

26   *Chapter 2. Financial instruments and pricing engines*

Listing 2.10: Implementation of the `VanillaOption` class.

```
VanillaOption::VanillaOption(
    const boost::shared_ptr<StrikedTypePayoff>& payoff,
    const boost::shared_ptr<Exercise>& exercise)
: payoff_(payoff), exercise_(exercise) {}

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

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

void VanillaOption::setupArguments(
                    PricingEngine::arguments* args) const {
    VanillaOption::arguments* arguments =
        dynamic_cast<VanillaOption::arguments*>(args);
    QL_REQUIRE(arguments != 0, "wrong argument type");
    arguments->exercise = exercise_;
    arguments->payoff = payoff_;
}

void VanillaOption::fetchResults(
                    const PricingEngine::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
}
```

contains market-data information about the underlying including present value, risk-free rate, dividend yield, and volatility. Once again, the actual calculations are hidden behind the interface of another class, namely, the `BlackCalculator` class. However, the code has enough detail to show a few relevant features.

The method starts by verifying a few preconditions. This might come as a surprise, since the arguments of the calculations are already validated by the time

Listing 2.11: Sketch of an engine for the VanillaOption class.

```
class AnalyticEuropeanEngine
    : public VanillaOption::engine {
  public:
    AnalyticEuropeanEngine(
      const shared_ptr<GeneralizedBlackScholesProcess>&
                                            process)
    : process_(process) {
        registerWith(process);
    }
    void calculate() const {
      QL_REQUIRE(
          arguments_.exercise->type() == Exercise::European,
          "not an European option");
      shared_ptr<PlainVanillaPayoff> payoff =
          dynamic_pointer_cast<PlainVanillaPayoff>(
                                    arguments_.payoff);
      QL_REQUIRE(process, "Black-Scholes process needed");
      ... // other requirements

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

      BlackCalculator black(payoff, forwardPrice,
                            stdDev, discount);

      results_.value = black.value();
      results_.delta = black.delta(spot);
      ... // other greeks
    }
  private:
    shared_ptr<GeneralizedBlackScholesProcess> process_;
};
```

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, one such requirement is that the option is European and that the payoff is a plain call/put one, which also means that the payoff will be cast down to the needed class.[9]

---

[9] boost::dynamic_pointer_cast is the equivalent of dynamic_cast for shared pointers.

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 forward price of the underlying and the risk-free discount factor at maturity, are also extracted[10].

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. Alternative designs

You might still be wondering whether the need for downcasting can be eliminated. It could; but it 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 [8]; 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 me to provide no real advantage, while making it more cumbersome for developers to create new instruments. Also, it makes it impossible to derive from a class once the template cycle is closed; this might prevent one, for instance, from coding both a bond class usable on its own and a derived fixed-coupon bond class. Therefore, it is likely that the current `Instrument` implementation will remain unchanged in this respect.

---

[10]The interested reader can find the full code of the engine in the QuantLib sources.

Listing 2.12: Sketch of an `Instrument` class template.

```
template <class A, class R>
class GenericEngine {
  public:
    A& getArguments() const { return arguments_; }
    const R& getResults() 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_->getArguments());
        engine_->arguments().validate();
        engine_->calculate();
        static_cast<const T*>(this)->fetchResults(
                              engine_->getResults());
    }
  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