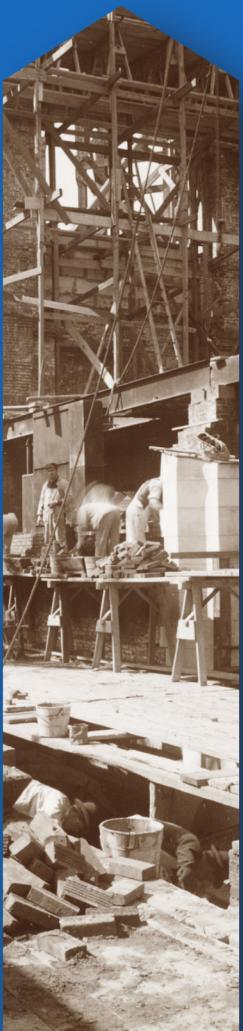


Luigi Ballabio

# IMPLEMENTING QUANTLIB

Quantitative finance in C++:  
an inside look at the architecture  
of the QuantLib library



# Implementing QuantLib

Luigi Ballabio

This book is for sale at <http://leanpub.com/implementingquantlib>

This version was published on 2021-01-16



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2014 - 2021 Luigi Ballabio

## **Tweet This Book!**

Please help Luigi Ballabio by spreading the word about this book on [Twitter](#)!

The suggested hashtag for this book is [#quantlib](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

[#quantlib](#)

## Also By Luigi Ballabio

QuantLib Python Cookbook

构建 QuantLib

Implementing QuantLib の和訳

# Contents

1.	<b>Introduction</b>	1
2.	<b>Financial instruments and pricing engines</b>	4
2.1	The <code>Instrument</code> class	4
2.1.1	Interface and requirements	4
2.1.2	Implementation	5
2.1.3	Example: interest-rate swap	9
2.1.4	Further developments	13
2.2	Pricing engines	13
2.2.1	Example: plain-vanilla option	20
3.	<b>Term structures</b>	26
3.1	The <code>TermStructure</code> class	26
3.1.1	Interface and requirements	26
3.1.2	Implementation	27
3.2	Interest-rate term structures	31
3.2.1	Interface and implementation	31
3.2.2	Discount, forward-rate, and zero-rate curves	33
3.2.3	Example: bootstrapping an interpolated curve	37
3.2.4	Example: adding z-spread to an interest-rate curve	47
3.3	Other term structures	49
3.3.1	Default-probability term structures	49
3.3.2	Inflation term structures	52
3.3.3	Volatility term structures	54
3.3.4	Equity volatility structures	55
3.3.5	Interest-rate volatility structures	58
4.	<b>Cash flows and coupons</b>	63
4.1	The <code>CashFlow</code> class	63
4.2	Interest-rate coupons	64
4.2.1	Fixed-rate coupons	66
4.2.2	Floating-rate coupons	68
4.2.3	Example: LIBOR coupons	73
4.2.4	Example: capped/floored coupons	78

## CONTENTS

4.2.5	Generating cash-flow sequences . . . . .	84
4.2.6	Other coupons and further developments . . . . .	87
4.3	Cash-flow analysis . . . . .	88
4.3.1	Example: fixed-rate bonds . . . . .	91
5.	<b>Parameterized models and calibration</b> . . . . .	98
5.1	The <code>CalibrationHelper</code> class . . . . .	98
5.1.1	Example: the Heston model . . . . .	103
5.2	Parameters . . . . .	105
5.3	The <code>CalibratedModel</code> class . . . . .	108
5.3.1	Example: the Heston model, continued . . . . .	114
6.	<b>The Monte Carlo framework</b> . . . . .	117
6.1	Path generation . . . . .	117
6.1.1	Random-number generation . . . . .	117
6.1.2	Stochastic processes . . . . .	121
6.1.3	Random path generators . . . . .	132
6.2	Pricing on a path . . . . .	138
6.3	Putting it all together . . . . .	139
6.3.1	Monte Carlo traits . . . . .	139
6.3.2	The Monte Carlo model . . . . .	142
6.3.3	Monte Carlo simulations . . . . .	145
6.3.4	Example: basket option . . . . .	149
7.	<b>The tree framework</b> . . . . .	155
7.1	The <code>Lattice</code> and <code>DiscretizedAsset</code> classes . . . . .	155
7.1.1	Example: discretized bonds . . . . .	161
7.1.2	Example: discretized option . . . . .	165
7.2	Trees and tree-based lattices . . . . .	168
7.2.1	The <code>Tree</code> class template . . . . .	168
7.2.2	Binomial and trinomial trees . . . . .	172
7.2.3	The <code>TreeLattice</code> class template . . . . .	182
7.3	Tree-based engines . . . . .	189
7.3.1	Example: callable fixed-rate bonds . . . . .	189
8.	<b>The finite-difference framework</b> . . . . .	195
8.1	The old framework . . . . .	195
8.1.1	Differential operators . . . . .	195
8.1.2	Evolution schemes . . . . .	198
8.1.3	Boundary conditions . . . . .	201
8.1.4	Step conditions . . . . .	205
8.1.5	The <code>FiniteDifferenceModel</code> class . . . . .	207
8.1.6	Example: American option . . . . .	212
8.1.7	Time-dependent operators . . . . .	220

## CONTENTS

8.2	The new framework . . . . .	224
8.2.1	Meshers . . . . .	226
8.2.2	Operators . . . . .	229
8.2.3	Examples: Black-Scholes operators . . . . .	238
8.2.4	Initial, boundary, and step conditions . . . . .	242
8.2.5	Schemes and solvers . . . . .	247
9.	Conclusion . . . . .	254
A.	Odds and ends . . . . .	255
Basic types . . . . .	255	
Date calculations . . . . .	256	
Dates and periods . . . . .	256	
Calendars . . . . .	257	
Day-count conventions . . . . .	259	
Schedules . . . . .	260	
Finance-related classes . . . . .	261	
Market quotes . . . . .	262	
Interest rates . . . . .	264	
Indexes . . . . .	266	
Exercises and payoffs . . . . .	272	
Math-related classes . . . . .	275	
Interpolations . . . . .	275	
One-dimensional solvers . . . . .	279	
Optimizers . . . . .	281	
Statistics . . . . .	285	
Linear algebra . . . . .	288	
Global settings . . . . .	291	
Utilities . . . . .	294	
Smart pointers and handles . . . . .	295	
Error reporting . . . . .	298	
Disposable objects . . . . .	300	
Design patterns . . . . .	303	
The Observer pattern . . . . .	303	
The Singleton pattern . . . . .	306	
The Visitor pattern . . . . .	307	
B.	Code conventions . . . . .	310
QuantLib license . . . . .	312	
Bibliography . . . . .	316	

The author has used good faith effort in preparation of this book, but makes no expressed or implied warranty of any kind and disclaims without limitation all responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein. Use of the information and instructions in this book is at your own risk.

The cover image is in the public domain and available from the [New York Public Library](#). The cover font is Open Sans Condensed, released by [Steve Matteson](#) under the [Apache License version 2.0](#).

# 1. Introduction

With the enthusiasm of youth, [the QuantLib web site](#) used to state that QuantLib aimed at becoming “the standard free/open-source financial library.” By interpreting such statement a bit loosely, one might say that it has somewhat succeeded—albeit by employing the rather devious trick of being the first, and thus for some time the *only* open-source financial library<sup>1</sup>.

Standard or not, the project is thriving; at the time of this writing, each new release is downloaded a few thousand times, there is a steady stream of contributions from users, and the library seems to be used in the real world—as far as I can guess through the usual shroud of secrecy used in the financial world. All in all, as a project administrator, I can declare myself a happy camper.

But all the more for that, the lack of proper documentation shows. Although a detailed class reference is available (that was an easy task, since it can be generated automatically) it doesn’t let one see the forest for the trees; so that a new user might get the impression that the QuantLib developers share the views of the Bellman from Lewis Carroll’s *Hunting of the Snark*:

“What use are Mercator’s North Poles and Equators,  
Tropics, Zones and Meridian Lines?”  
So the Bellman would cry: and the crew would reply,  
“They are merely conventional signs!”

The purpose of this book is to fill a part of the existing void. It is a report on the design and implementation of QuantLib, alike in spirit—but, hopefully, with less frightening results—to the *How I did it* book<sup>2</sup> prominently featured in Mel Brooks’ *Young Frankenstein*. If you are—or want to be—a QuantLib user, you will find here useful information on the design of the library that might not be readily apparent when reading the code. If you’re working in quantitative finance, even if not using QuantLib, you can still read it as a field report on the design of a financial library. You will find that it covers issues that you might also face, as well as some possible solutions and their rationale. Based on your constraints, it is possible—even likely—that you will choose other solutions; but you might profit from this discussion just the same.

In my descriptions, I’ll also point out shortcomings in the current implementation; not to disparage the library (I’m pretty much involved in it, after all) but for more useful purposes. On the one hand, describing the existing pitfalls will help developers avoid them; on the other hand, it might show how to improve the library. Indeed, it already happened that reviewing the code for this book caused me to go back and modify it for the better.

---

<sup>1</sup>A few gentle users happened to refer to our library as “*the QuantLib*.” As much as I like the expression, modesty and habit have so far prevented me from using it.

<sup>2</sup>In this case, of course, it would be “*How we did it*.”

For reasons of both space and time, I won't be able to cover every aspect of the library. In the first half of the book, I'll describe a few of the most important classes, such as those modeling financial instruments and term structures; this will give you a view of the larger architecture of the library. In the second half, I'll describe a few specialized frameworks, such as those used for creating Monte Carlo or finite-differences models. Some of those are more polished than others; I hope that their current shortcomings will be as interesting as their strong points.

The book is primarily aimed at users wanting to extend the library with their own instruments or models; if you desire to do so, the description of the available class hierarchies and frameworks will provide you with information about the hooks you need to integrate your code with QuantLib and take advantage of its facilities. If you're not this kind of user, don't close the book yet; you can find useful information too. However, you might want to look at the *QuantLib Python Cookbook* instead. It can be useful to C++ users, too.

\* \* \*

And now, as is tradition, a few notes on the style and requirements of this book.

Knowledge of both C++ and quantitative finance is assumed. I've no pretense of being able to teach you either one, and this book is thick enough already. Here, I just describe the implementation and design of QuantLib; I'll leave it to other and better authors to describe the problem domain on one hand, and the language syntax and tricks on the other.

As you already noticed, I'll write in the first person singular. True, it might look rather self-centered—as a matter of fact, I hope you still haven't put down the book in annoyance—but we would feel rather pompous if we were to use the first person plural. The author of this book feels the same about using the third person. After a bit of thinking, I opted for a less formal but more comfortable style (which, as you noted, also includes a liberal use of contractions). For the same reason, I'll be addressing *you* instead of the proverbial acute reader. The use of the singular will also help to avoid confusion; when I use the plural, I describe work done by the QuantLib developers as a group.

I will describe the evolution of a design when it is interesting on its own, or relevant for the final result. For the sake of clarity, in most cases I'll skip over the blind alleys and wrong turns taken; put together design decisions which were made at different times; and only show the final design, sometimes simplified. This will still leave me with plenty to say: in the words of the Alabama Shakes, “why” is an awful lot of question.

I will point out the use of design patterns in the code I describe. Mind you, I'm not advocating cramming your code with them; they should be applied when they're useful, not for their own sake.<sup>3</sup> However, QuantLib is now the result of several years of coding and refactoring, both based on user feedback and new requirements being added over time. It is only natural that the design evolved toward patterns.

---

<sup>3</sup>A more thorough exposition of this point can be found in [Kerievsky, 2004](#).

I will apply to the code listings in the book the same conventions used in the library and outlined in [appendix B](#). I will depart from them in one respect: due to the limitations on line length, I might drop the std and boost namespaces from type names. When the listings need to be complemented by diagrams, I will be using UML; for those not familiar with this language, a concise guide can be found in [Fowler, 2003](#).

\* \* \*

And now, let's dive.

# 2. Financial instruments and pricing engines

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 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.

The idea, of course, is to be able to write code such as

```
for (i = portfolio.begin(); i != portfolio.end(); ++i)
    totalNPV += i->NPV();
```

where we don't have to care about the specific type of each instrument. However, this also prevents us from knowing what arguments to pass to the NPV method, or even what methods to call. Therefore, even the two seemingly harmless lines above tell us that we have to step back and think a bit about the interface.

### 2.1.1 Interface and requirements

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; since we can't specify what arguments are needed,<sup>1</sup> the methods take none; any needed input will have to be stored by the instrument. The resulting interface is shown in the listing below.

---

<sup>1</sup>Even fancy new C++11 stuff like variadic templates won't help.

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;
};
```

---

As is good practice, the methods were first 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, you might 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 that 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.

We were also concerned with 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 a simple 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 ([Gamma et al, 1995](#)). The pattern itself is briefly described<sup>2</sup> in appendix A; I describe here the participants.

---

<sup>2</sup>This does not excuse you from reading the Gang of Four book.

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.<sup>3</sup>

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. 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 the following listing.

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;
            }
        }
    }
}
```

---

<sup>3</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.

```

    }
};
```

---

The code is not overly complex. 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 ([Gamma et al, 1995](#)), sometimes also called *non-virtual interface*. 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 might wonder 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, we could have written the body of the algorithm more simply—for instance, as in the following, seemingly equivalent code, that doesn't catch and rethrow 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. If you're interested, you can heed 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.

Also, it should be noted that the C++11 standard now requires `const` methods to be thread-safe; that is, two threads calling `const` members at the same time should not incur in race conditions (to learn all about it, see [Sutter, 2013](#)). To make the code conform to the new standard, we should protect updates to `mutable` members with a mutex. This would likely require some changes in design.

The `Instrument` class inherits from `LazyObject`. In order to implement the interface outlined earlier, it decorates the `calculate` method with code specific to financial instruments. The resulting method is shown in the listing below, together with other bits of supporting code.

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;
    }
};
```

---

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>4</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 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 you surely know, 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>5</sup> as a new class deriving from `Instrument`. Its outline is shown in the following listing.

Partial interface of the `Swap` class.

---

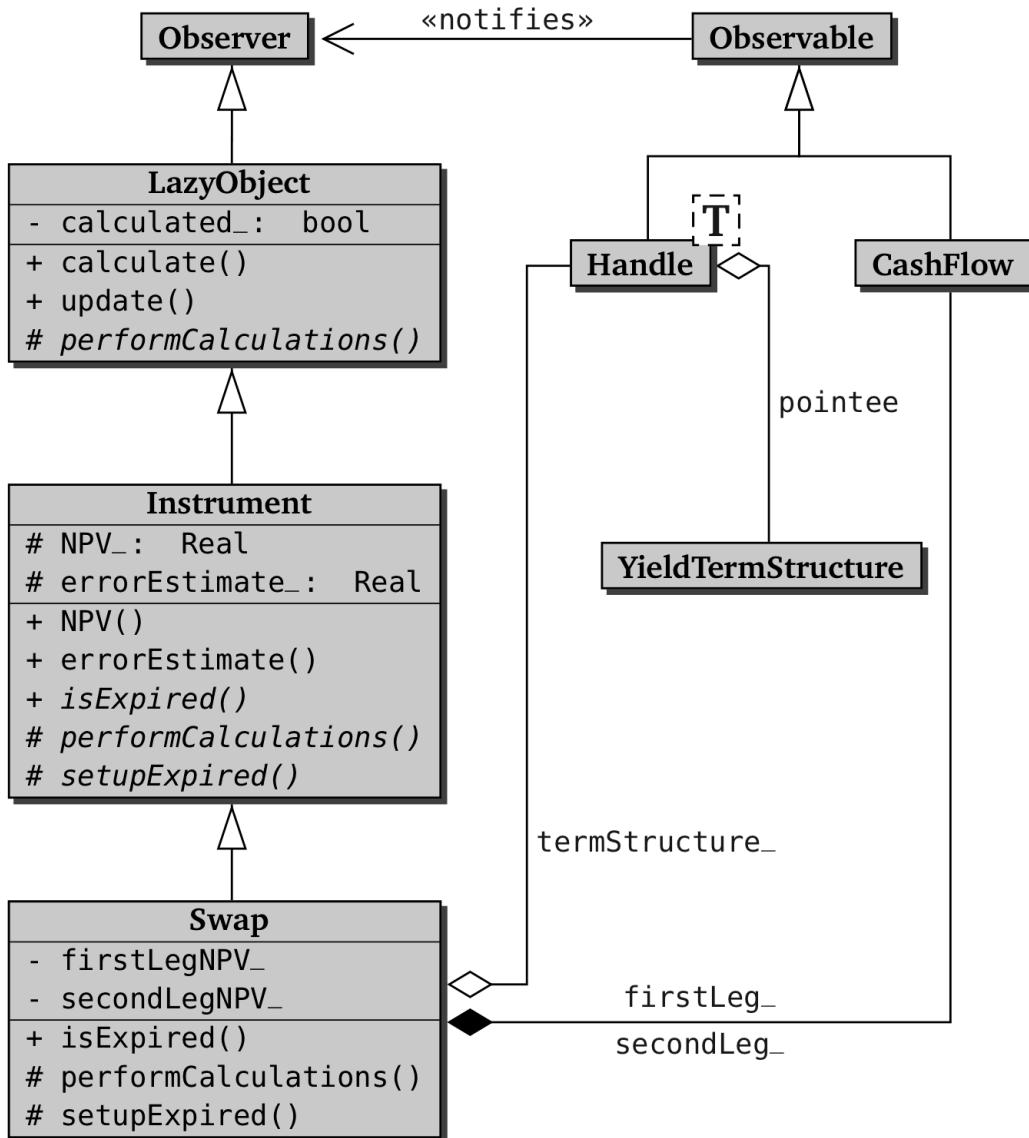
```
class Swap : public Instrument {
public:
    Swap(const vector<shared_ptr<CashFlow>>& firstLeg,
          const vector<shared_ptr<CashFlow>>& secondLeg,
          const Handle<YieldTermStructure>& termStructure);
    bool isExpired() const;
    Real firstLegBPS() const;
    Real secondLegBPS() const;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

---

<sup>4</sup>The `Instrument` class also defines an `errorEstimate_` member, which is omitted here for clarity of exposition. The discussion of `NPV_` applies to both.

<sup>5</sup>The implementation shown in this section is somewhat outdated. However, I'm still using it here since it provides a simpler example.

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 the figure below.



Class diagram of the `Swap` class.

The fitting of the class to the `Instrument` framework is done in three steps, the third being optional

depending on the derived class; the relevant methods are shown in the next listing.

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;
}

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 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.

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.

### 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 means of the handle) whereas the cash flows are part of the definition of the swap and are thus immutable.

The last required method is `performCalculations`. The calculation is performed by calling two external functions from the `Cashflows` class.<sup>6</sup> 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. 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.<sup>7</sup>

The third and final step only needs to be performed if—as in this case—the class defines additional

<sup>6</sup>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 the calculations. Your curiosity will be satisfied in a later chapter devoted to cash flows and related functions.

<sup>7</sup>`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`.

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

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 you wanted to add the values of two instruments whose values are not in the same currency; you would have to convert manually one of the values to the currency of the other before adding them together.

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 might 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.

However, this would be a major change, affecting a large part of the code base in a number of ways. Therefore, it will need some serious thinking before we tackle it (if we do tackle it at all).

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.<sup>8</sup> The means to do this are the subject of the next section.

## 2.2 Pricing engines

We now turn to the second of the requirements 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. Let's take the classic textbook example—the European equity option. One might want to price it by means of the analytic Black-Scholes formula in order to

---

<sup>8</sup>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.

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 won't work as such. The problem is, we want to implement the dispatching code just once, namely, in the `Instrument` class. However, that class doesn't know the number and type of arguments; different derived 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 explicit arguments to the engine through a method, as the one outlined above, would thus lead to undesirable consequences. Pricing engines for different instruments would have different interfaces, which would prevent us from defining a single base class; therefore, 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.

The listing below shows the interface of the resulting `PricingEngine` class, as well as its inner `arguments` 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. 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, while `results::reset` is to be called before the engine starts calculating in order to clean previous results.

Interface of `PricingEngine` and of related classes.

---

```

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_;
};
```

---

Armed with our new classes, we can now write 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 the next listing. Note that an inner class `Instrument::result` was defined; it inherits from `PricingEngine::results` and contains the results that have to be provided for any instrument<sup>9</sup>

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:
        shared_ptr<PricingEngine> engine_;
};
```

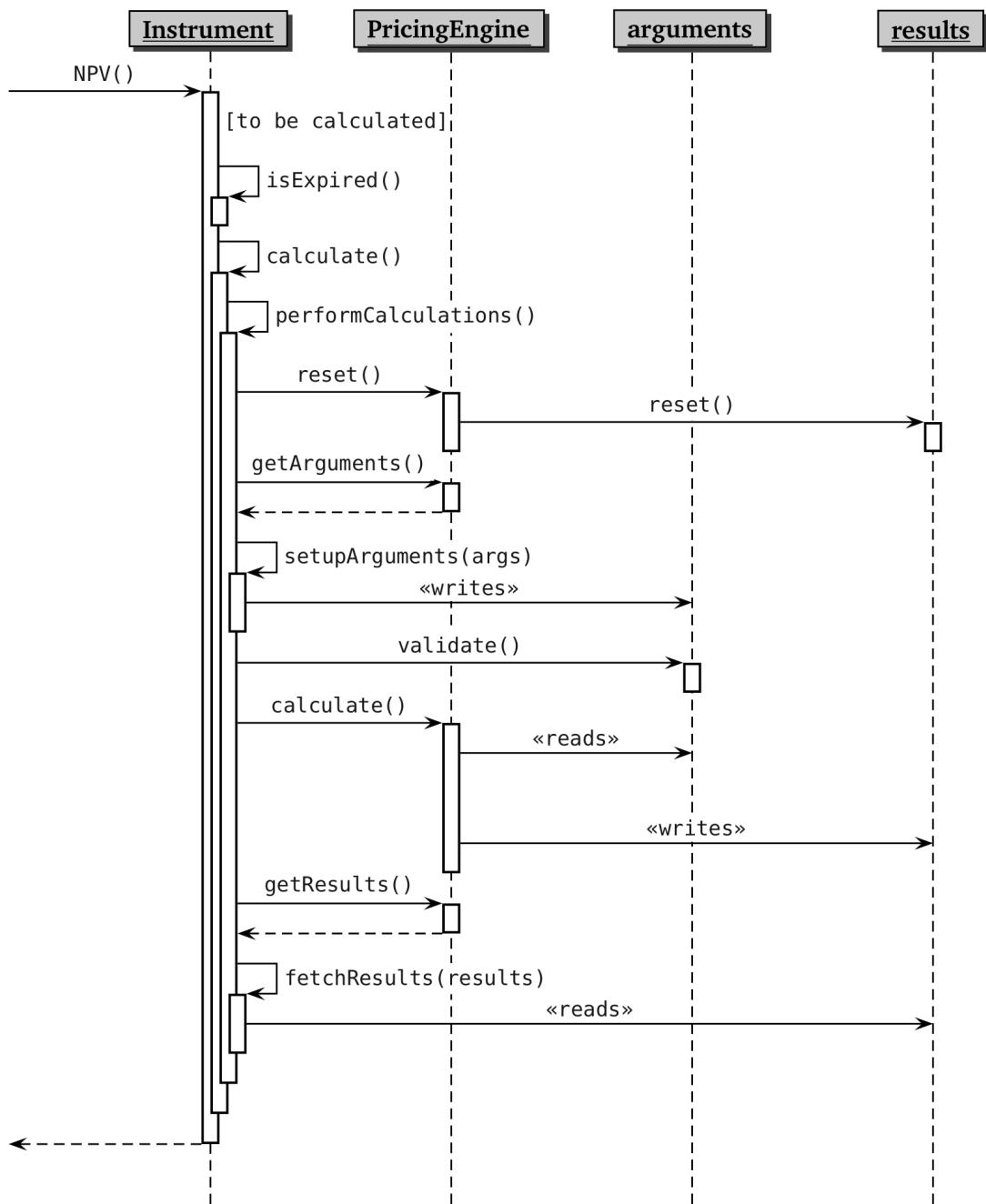
---

<sup>9</sup>The `Instrument::results` class also contains a `std::map` where pricing engines can store additional results. The relevant code is here omitted for clarity.

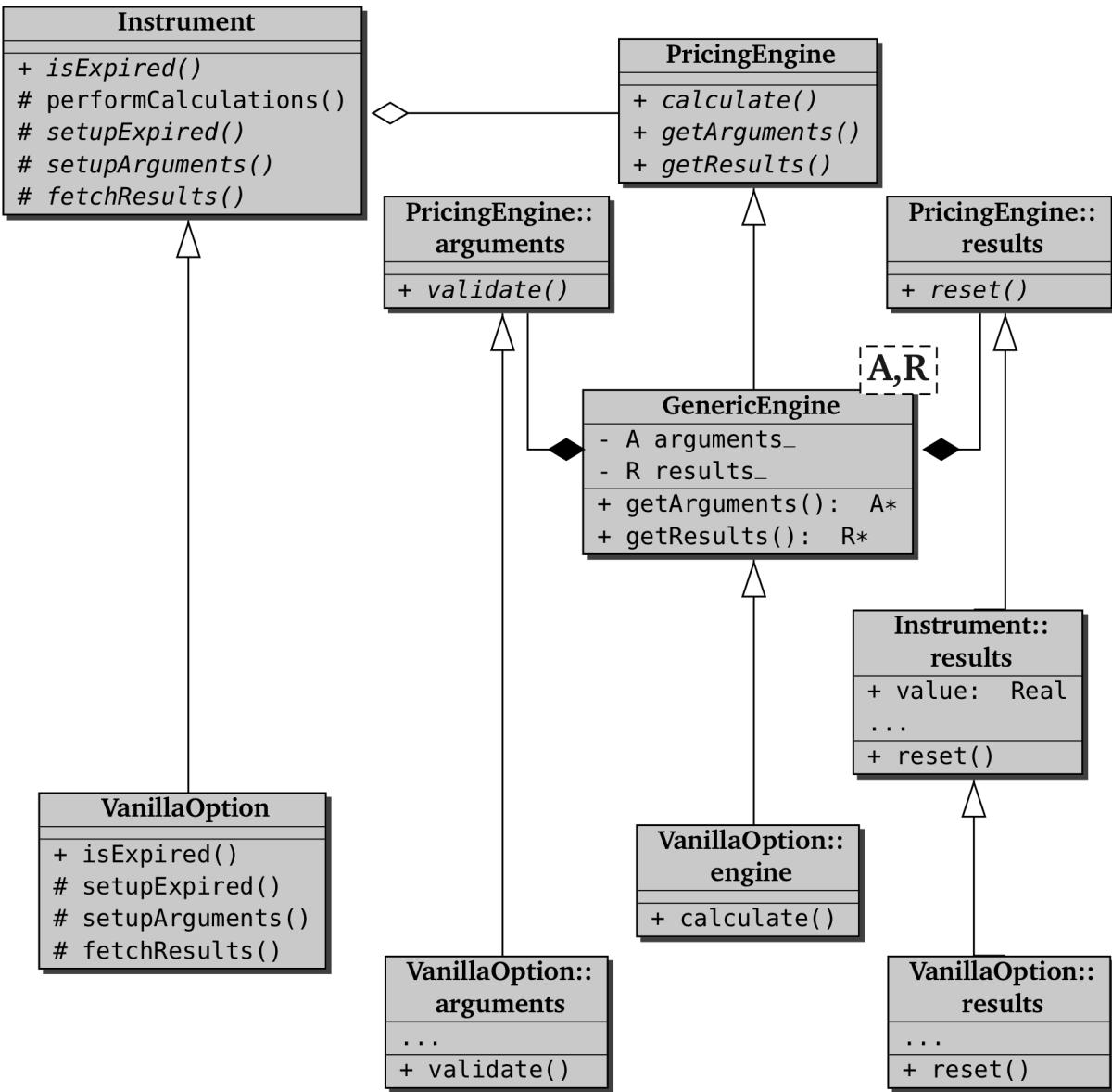
```
class Instrument::results
    : public virtual PricingEngine::results {
public:
    Value() { reset(); }
    void reset() {
        value = errorEstimate = Null<Real>();
    }
    Real value;
    Real errorEstimate;
};
```

---

As for `performCalculation`, 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 the first of the next two figures; the static relationship between the classes (and a possible concrete instrument) is shown in the second.



Sequence diagram of the interplay between instruments and pricing engines.



Class diagram of **Instrument**, **PricingEngine**, and related classes including a derived instrument class.

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 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 the 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.

### Aside: impure virtual methods.

Upon looking at the final implementation of the `Instrument` class, 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 meaningless method, were they to decide that some of their classes should simply override the `performCalculation` method.

## 2.2.1 Example: plain-vanilla option

At this point, an example is necessary. A word of warning, though: 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 deep 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; 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 would not make for an extremely clear example. Therefore, I will describe a simplified `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 as if it were implemented in the example class rather than inherited.

The listing below shows the 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.

Interface of the `VanillaOption` class.

---

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

---

Besides its own data and methods, `VanillaOption` declares a number of accessory classes: that is, the specific argument and result structures and a base pricing engine. They are defined as inner classes to highlight the relationship between them and the option class; their interface is shown in the next listing.

Interface of the `VanillaOption` inner classes.

---

```

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

```

---

Two comments can be made on such accessory classes. The first is that, making an exception to what I said in my introduction to the example, I didn't declare all data members into the `results` class. This was done in order to point out an implementation detail. One might want 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 `Instrument::results` to obtain the final structure. In this case, virtual inheritance from `PricingEngine::results` must be used to avoid the infamous inheritance diamond (see, for instance, [Stroustrup, 2013](#); the name is in the index).

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 the following listing.

#### Implementation of the `VanillaOption` class.

---

```

VanillaOption::VanillaOption(
    const shared_ptr<StrikedTypePayoff>& payoff,
    const 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
}

Real VanillaOption::delta() const {
    calculate();
    QL_ENSURE(delta_ != Null<Real>(), "delta not given");
    return delta_;
}

```

---

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 type (i.e., call or put) of the option, and the exercise contains information on the exercise dates and variety (i.e., European, American, or Bermudan). The passed arguments are stored in the corresponding data members. Also, note that they do not include market data; those will be passed elsewhere.

The methods related to expiration are straightforward; `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 calculations (say, converting dates into times) 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 copies the results into his own data members.

Any method that returns additional results, such as `delta`, will do as `NPV` does: it will call `calculate`, check that the corresponding result was cached (because any given engine might or might not be able to calculate it) and return it.

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 the listing below and implements the analytic Black-Scholes-Merton formula for European options.

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_;
};
```

---

Its constructor takes (and registers itself with) a Black-Scholes stochastic process that 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 were 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, 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.<sup>10</sup>

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.<sup>11</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.

<sup>10</sup>`dynamic_pointer_cast` is the equivalent of `dynamic_cast` for shared pointers.

<sup>11</sup>You can find the full code of the engine in the QuantLib sources.

# 3. Term structures

Change is the only constant, as Heraclitus said. Paradoxically, the aphorism still holds after twenty-five centuries; also in quantitative finance, where practically all quantities obviously vary—sometimes spectacularly—over time.

This leads us straight to the subject of term structures. This chapter describes the basic facilities available for their construction, as well as a few existing term structures that can be used as provided.

## 3.1 The `TermStructure` class

The current base class for term structures is a fine example of design *ex-post*. After some thinking, you might come up with a specification for such class. When we started the library, we didn't; we just started growing classes as we needed them. A couple of years later, older and somewhat wiser, we looked at the existing term structures and abstracted out their common features. The result is the `TermStructure` class as described in this section.

### 3.1.1 Interface and requirements

Once abstracted out, the base term-structure class (whose interface is shown in the listing below) was responsible for three basic tasks.

Interface of the `TermStructure` class.

---

```
class TermStructure : public virtual Observer,
                     public virtual Observable,
                     public Extrapolator {

public:
    TermStructure(const DayCounter& dc = DayCounter());
    TermStructure(const Date& referenceDate,
                  const Calendar& calendar = Calendar(),
                  const DayCounter& dc = DayCounter());
    TermStructure(Natural settlementDays,
                  const Calendar&,
                  const DayCounter& dc = DayCounter());
    virtual ~TermStructure();

    virtual DayCounter dayCounter() const;
    virtual Date maxDate() const = 0;
```

```

    virtual Time maxTime() const;
    virtual const Date& referenceDate() const;
    virtual Calendar calendar() const;
    virtual Natural settlementDays() const;
    Time timeFromReference(const Date& date) const;

    void update();
protected:
    void checkRange(const Date&, bool extrapolate) const;
    void checkRange(Time, bool extrapolate) const;

    bool moving_;
};
```

---

The first is to keep track of its own reference date, i.e., the date at which—in a manner of speaking—the future begins.<sup>1</sup> For a volatility term structure, that would most likely be today’s date. For a yield curve, it might be today’s date, too; but depending on the conventions used at one’s desk (for instance, an interest-rate swap desk whose deals are all settled spot; that’s on the second next business day for you equity folks) the reference date might be the result of advancing today’s date by a few business days. Our term-structure class must be able to perform such a calculation if needed. Also, there might be cases in which the reference date is specified externally (such as when a sequence of dates, including the reference, is tabulated somewhere together with the corresponding discount factors). Finally, the calculation of the reference date might be altogether delegated to some other object; we’ll see such an arrangement in [a later section](#). In all these cases, the reference date will be provided to client code by means of the `referenceDate` method. The related `calendar` and `settlementDays` methods return the calendar and the number of days used for the calculation (“settlement” applies to instruments, but is probably not the correct word for a term structure).

The second (and somewhat mundane) task is to convert dates to times, i.e., points on a real-valued time axis starting with  $t = 0$  at the reference date. Such times might be used in the mathematical model underlying the curve, or simply to convert, say, from discount factors to zero-yield rates. The calculation is made available by means of the `timeFromReference` method.

The third task (also a mundane one) is to check whether a given date or time belongs to the domain covered by the term structure. The `TermStructure` class delegates to derived classes the specification of the latest date in the domain—which must be implemented in the `maxDate` method—and provides a corresponding `maxTime` method as well as an overloaded `checkRange` method performing the actual test; there is no `minDate` method, as the domain is assumed to start at the reference date.

### 3.1.2 Implementation

---

<sup>1</sup>This is not strictly true of all term structures. However, we’ll leave it at that for the time being.

---

**Implementation of the TermStructure class.**


---

```

TermStructure::TermStructure(const DayCounter& dc)
: moving_(false), updated_(true),
  settlementDays_(Null<Natural>()), dayCounter_(dc) {}

TermStructure::TermStructure(const Date& referenceDate,
                           const Calendar& calendar,
                           const DayCounter& dc)
: moving_(false), referenceDate_(referenceDate),
  updated_(true), settlementDays_(Null<Natural>()),
  calendar_(calendar), dayCounter_(dc) {}

TermStructure::TermStructure(Natural settlementDays,
                           const Calendar& calendar,
                           const DayCounter& dc)
: moving_(true), updated_(false),
  settlementDays_(settlementDays),
  calendar_(calendar), dayCounter_(dc) {
    registerWith(Settings::instance().evaluationDate());
}

DayCounter TermStructure::dayCounter() const {
    return dayCounter_;
}

Time TermStructure::maxTime() const {
    return timeFromReference(maxDate());
}

const Date& TermStructure::referenceDate() const {
    if (!updated_) {
        Date today = Settings::instance().evaluationDate();
        referenceDate_ =
            calendar().advance(today, settlementDays_, Days);
        updated_ = true;
    }
    return referenceDate_;
}

Calendar TermStructure::calendar() const {
    return calendar_;
}

```

```

Natural TermStructure::settlementDays() const {
    return settlementDays_;
}

Time TermStructure::timeFromReference(const Date& d) const {
    return dayCounter().yearFraction(referenceDate(),d);
}

void TermStructure::update() {
    if (moving_)
        updated_ = false;
    notifyObservers();
}

void TermStructure::checkRange(const Date& d,
                               bool extrapolate) const {
    checkRange(timeFromReference(d),extrapolate);
}

void TermStructure::checkRange(Time t,
                               bool extrapolate) const {
    QL_REQUIRE(t >= 0.0,
               "negative time (" << t << ") given");
    QL_REQUIRE(extrapolate || allowsExtrapolation()
               || t <= maxTime(),
               "time (" << t
               << ") is past max curve time (" << maxTime() << ")");
}

```

---

The first task—keeping track of the reference date—starts when the term structure is instantiated. Depending on how the reference date is to be calculated,<sup>2</sup> different constructors must be called. All such constructors set two boolean data members. The first is called `moving_`; it is set to `true` if the reference date moves when today's date changes, or to `false` if the date is fixed. The second, `updated_`, specifies whether the value of another data member (`referenceDate_`, storing the latest calculated value of the reference date) is currently up to date or should be recalculated.

Three constructors are available. One simply takes a day counter (used for time calculations, as we will see later) but no arguments related to reference-date calculation. Of course, the resulting term structure can't calculate such date; therefore, derived classes calling this constructor must take care of the calculation by overriding the virtual `referenceDate` method. The implementation sets

<sup>2</sup>Before reading this section, you might want to skim briefly through [appendix A](#) for a description of the classes used for date- and time-related calculations.

`moving_` to `false` and `updated_` to `true` to inhibit calculations in the base class.

Another constructor takes a date, as well as an optional calendar and a day counter. When this one is used, the reference date is assumed to be fixed and equal to the given date. Accordingly, the implementation sets `referenceDate_` to the passed date, `moving_` to `false`, and `updated_` to `true`.

Finally, a third constructor takes a number of days and a calendar. When this one is used, the reference date will be calculated as today's date advanced by the given number of business days according to the given calendar. Besides copying the passed data to the corresponding data members, the implementation sets `moving_` to `true` and `updated_` to `false` (since no calculation is performed at this time). However, that's not the full story; if today's date changes, the term structure must be notified so that it can update its reference date. The `Settings` class (described in [appendix A](#)) provides global access to the current evaluation date, with which the term structure registers as an observer. When a change is notified, the `update` method is executed. If the reference date is moving, the body of the method sets `updated_` to `false` before forwarding the notification to the term structure's own observers.

Apart from trivial inspectors such as the `calendar` method, the implementation of the first task is completed with the `referenceDate` method. If the reference date needs to be calculated, it does so by retrieving the current evaluation date, advancing it as specified, and storing the result in the `referenceDate_` data member before returning it.

The second task is much simpler, since the conversion of dates into times can be delegated entirely to a `DayCounter` instance. Such day counter is usually passed to the term structure as a constructor argument and stored in the `dayCounter_` data member. The conversion is handled by the `timeFromReference` method, which asks the day counter for the number of years between the reference date and the passed date. Note that, in the body of the method, both the day counter and the reference date are accessed by means of the corresponding methods rather than the data members. This is necessary, since—as I mentioned earlier—the `referenceDate` method can be overridden entirely and thus disregard the data member; the same applies to the `dayCounter` method.

You might object that this is, to use the term coined by Kent Beck ([Fowler et al, 1999](#)), a code smell. A term-structure instance might store a day counter or a reference date (or likely both) that don't correspond to the actual ones used by its methods. This disturbs me as well; and indeed, earlier versions of the class declared the `dayCounter` method as purely virtual and did not include the data member. However, it is a necessary evil in the case of the reference date, since we need a data member to cache its calculated value. Due to the broken-window effect ([Hunt and Thomas, 1999](#)), the day counter, calendar and settlement days followed (after a period in which we developed a number of derived term structures, all of which had to define the same data members).

What day counter should be used for a given term structure? Fortunately, it doesn't matter much. If one is only working with dates (i.e., provides dates as an input for the construction of the term structure and uses dates as arguments to retrieve values) the effects of choosing a specific day counter will cancel out as long as the day counter is sufficiently well behaved: for instance, if it is homogeneous (by which I mean that the time  $T(d_1, d_2)$  between two dates  $d_1$  and  $d_2$  equals the time  $T(d_3, d_4)$  between  $d_3$  and  $d_4$  if the two pairs of dates differ by the same number of days) and

additive (by which I mean that  $T(d_1, d_2) + T(d_2, d_3)$  equals  $T(d_1, d_3)$  for all choices of the three dates). Two such day counters are the actual/360 and the actual/365-fixed ones. Similarly, if one is only working with times, the day counter will not be used at all.

Onwards with the third and final task. The job of defining the valid date range is delegated to derived classes, which must define the `maxDate` method (here declared as purely virtual). The corresponding time range is calculated by the `maxTime` method, which converts the latest valid date to time by means of the `timeFromReference` method; this, too, can be overridden. Finally, the two `checkRange` methods implement the actual range checking and throw an exception if the passed argument is not in the valid range; the one that takes a date does so by forwarding the request to the other after converting the given date to a time. The check can be overridden by a request to extrapolate outside the domain of the term-structure; this can be done either by passing an optional boolean argument to `checkRange` or by using the facilities provided by the `Extrapolator` class (see appendix A) from which `TermStructure` inherits. Extrapolation is only allowed beyond the maximum date; requests for dates before the reference date are always rejected.

### Aside: evaluation date tricks.

If no evaluation date is set, the `Settings` class defaults to returning today's date. Unfortunately, the latter will change silently (that is, without notifying its observers) at the strike of midnight, causing mysterious errors. If you run overnight calculations, you'll have to perform the same feat as Hiro Nakamura in *Heroes*—freeze time. Explicitly settings today's date as the evaluation date will keep it fixed, even when today becomes tomorrow.

Another trick worth knowing: if all your term structures are moving, setting the evaluation date to tomorrow and recalculating the value of your instruments while keeping everything else unchanged will give you the daily theta of your portfolio.

## 3.2 Interest-rate term structures

The `YieldTermStructure` class predates `TermStructure`—in fact, it was even called `TermStructure` back in the day, when it was the only kind of term structure in the library and we still hadn't seen the world. Its interface provides the means to forecast interest rates and discount factors at any date in the curve domain; also, it implements some machinery to ease the task of writing a concrete yield curve.

### 3.2.1 Interface and implementation

The interface of the `YieldTermStructure` class is sketched in the following listing.

Partial interface of the `YieldTermStructure` class.

---

```

class YieldTermStructure : public TermStructure {
public:
    YieldTermStructure(const DayCounter& dc = DayCounter());
    YieldTermStructure(const Date& referenceDate,
                      const Calendar& cal = Calendar(),
                      const DayCounter& dc = DayCounter());
    YieldTermStructure(Natural settlementDays,
                      const Calendar&,
                      const DayCounter& dc = DayCounter());

    InterestRate zeroRate(const Date& d,
                          const DayCounter& dayCounter,
                          Compounding compounding,
                          Frequency frequency = Annual,
                          bool extrapolate = false) const;

    InterestRate zeroRate(Time t,
                          Compounding compounding,
                          Frequency frequency = Annual,
                          bool extrapolate = false) const;

    DiscountFactor discount(const Date&,
                           bool extrapolate = false) const;
    // same at time t

    InterestRate forwardRate(const Date& d1,
                            const Date& d2,
                            const DayCounter& dayCounter,
                            Compounding compounding,
                            Frequency frequency = Annual,
                            bool extrapolate = false) const;
    // same starting from date d and spanning a period p
    // same between times t1 and t2

    // ...more methods
protected:
    virtual DiscountFactor discountImpl(Time) const = 0;
};

```

---

The constructors forward their arguments to the corresponding constructors in the `TermStructure` class—nothing to write home about. The other methods return information on the yield structure in

different ways: on the one hand, they can return zero rates, forward rates, and discount factors;<sup>3</sup> on the other hand, they are overloaded so that they can return information as function of either dates or times.

Of course, there is a relationship between zero rates, forward rates, and discount factors; the knowledge of any one of them is sufficient to deduce the others. (I won't bore you with the formulas here—you know them.) This is reflected in the implementation, outlined in the next listing; the Template Method pattern is used to implement all public methods directly or indirectly in terms of the protected `discountImpl` abstract method. Derived classes only need to implement the latter in order to return any of the above quantities.

Partial implementation of the `YieldTermStructure` class.

---

```
InterestRate YieldTermStructure::zeroRate(
    const Date& d,
    const DayCounter& dayCounter,
    Compounding comp,
    Frequency freq,
    bool extrapolate) const {
    // checks and/or special cases
    Real compound = 1.0/discount(d, extrapolate);
    return InterestRate::impliedRate(compound,
        referenceDate(), d,
        dayCounter, comp, freq);
}

DiscountFactor YieldTermStructure::discount(
    const Date& d,
    bool extrapolate) const {
    checkRange(d, extrapolate);
    return discountImpl(timeFromReference(d));
}
```

---

### 3.2.2 Discount, forward-rate, and zero-rate curves

What if the author of a derived class doesn't want to implement `discountImpl`, though? After all, one might want to describe a yield curve in terms, say, of zero rates. Ever ready to serve (just like Jeeves in the P.~G.~Wodehouse novels—not that you're Bernie Wooster, of course) QuantLib provides a couple of classes to be used in this case. The two classes (outlined in the listing below) are called `ZeroYieldStructure` and `ForwardRateStructure`. They use the Adapter pattern<sup>4</sup> to transform the discount-based interface of `YieldTermStructure` into interfaces based on zero-yield and instantaneous-forward rates, respectively.

<sup>3</sup>Rates are returned as instances of the `InterestRate` class, described briefly in [appendix A](#).

<sup>4</sup>In case you're keeping count, this would be another notch in the spine of our Gang-of-Four book.

Outline of the `ZeroYieldStructure` and `ForwardRateStructure` classes.

---

```

class ZeroYieldStructure : public YieldTermStructure {
    public:
        // forwarding constructors, not shown
    protected:
        virtual Rate zeroYieldImpl(Time) const = 0;
        DiscountFactor discountImpl(Time t) const {
            Rate r = zeroYieldImpl(t);
            return std::exp(-r*t);
        }
};

class ForwardRateStructure : public YieldTermStructure {
    public:
        // forwarding constructors, not shown
    protected:
        virtual Rate forwardImpl(Time) const = 0;
        virtual Rate zeroYieldImpl(Time t) const {
            // averages forwardImpl between 0 and t
        }
        DiscountFactor discountImpl(Time t) const {
            Rate r = zeroYieldImpl(t);
            return std::exp(-r*t);
        }
};

```

---

The implementation of `ZeroYieldStructure` is not complex. A few constructors (not shown here) forward their arguments to the corresponding constructors in the parent `YieldTermStructure` class. The Adapter pattern is implemented in the protected section: an abstract `zeroYieldImpl` method is declared and used to implement the `discountImpl` method. Thus, authors of derived classes only need to provide an implementation of `zeroYieldImpl` to obtain a fully functional yield curve.<sup>5</sup> Note that, due to the formula used to obtain the discount factor, such method must return zero yields as continuously-compounded annualized rates.

In a similar way, the `ForwardRateStructure` class provides the means to describe the curve in terms of instantaneous forward rates (again, on an annual basis) by implementing a `forwardImpl` method in derived classes. However, it has an added twist. In order to obtain the discount at a given time  $T$ , we have to average the instantaneous forwards between 0 and  $T$ , thus retrieving the corresponding zero-yield rate. This class can't make any assumption on the shape of the forwards; therefore, all it can do is to perform a numerical integration—an expensive calculation. In order to provide a hook for optimization, the average is performed in a virtual `zeroYieldImpl` method

---

<sup>5</sup>Of course, the other required methods (such as `maxDate`) must be implemented as well.

that can be overridden if a faster calculation is available. You might object that if an expression is available for the zero yields, one can inherit from `ZeroYieldStructure` and be done with it; however, it is conceptually cleaner to express the curve in terms of the forwards if they were the actual focus of the model.

The two adapter classes I just described and the base `YieldTermStructure` class itself were used to implement interpolated discount, zero-yield, and forward curves. The next listing outlines the `InterpolatedZeroCurve` class template; the other two (`InterpolatedForwardCurve` and `InterpolatedDiscountCurve`) are implemented in the same way.

Outline of the `InterpolatedZeroCurve` class template.

---

```

template <class Interpolator>
class InterpolatedZeroCurve : public ZeroYieldStructure {
    public:
        // constructor
        InterpolatedZeroCurve(
            const std::vector<Date>& dates,
            const std::vector<Rate>& yields,
            const DayCounter& dayCounter,
            const Interpolator& interpolator
                = Interpolator())
        : ZeroYieldStructure(dates.front(), Calendar(),
            dayCounter),
            dates_(dates), yields_(yields),
            interpolator_(interpolator) {
            // check that dates are sorted, that there are
            // as many rates as dates, etc.

            // convert dates_ into times_

            interpolation_ =
                interpolator_.interpolate(times_.begin(),
                    times_.end(),
                    data_.begin());
        }
        Date maxDate() const {
            return dates_.back();
        }
        // other inspectors, not shown
    protected:
        // other constructors, not shown
        Rate zeroYieldImpl(Time t) const {
            return interpolation_(t, true);
        }
    }
```

```

mutable std::vector<Date> dates_;
mutable std::vector<Time> times_;
mutable std::vector<Rate> data_;
mutable Interpolation interpolation_;
Interpolator interpolator_;
};
```

---

The template argument `Interpolator` has a twofold task. On the one hand, it acts as a traits class ([Myers, 1995](#)). It specifies the kind of interpolation to be used as well as a few of its properties, namely, how many points are required (e.g., at least two for a linear interpolation) and whether the chosen interpolation is global (i.e., whether or not moving a data point changes the interpolation in intervals that do not contain such point; this is the case, e.g., for cubic splines). On the other hand, it doubles as a poor man's factory; when given a set of data points, it is able to build and return the corresponding `Interpolation` instance.<sup>6</sup>

The public constructor takes the data needed to build the curve: the set of dates over which to interpolate, the corresponding zero yields, the day counter to be used, and an optional `Interpolator` instance. For most interpolations, the last parameter is not needed; it can be passed when the interpolation needs parameters. The implementation forwards to the parent `ZeroYieldStructure` class the first of the passed dates, assumed to be the reference date for the curve, and the day counter; the other arguments are stored in the corresponding data members. After performing a few consistency checks, it converts the dates into times (using, of course, the passed reference date and day counter), asks the interpolator to create an `Interpolation` instance, and stores the result.

At this point, the curve is ready to be used. The other required methods can be implemented as one-liners; `maxDate` returns the latest of the passed dates, and `zeroYieldImpl` returns the interpolated value of the zero yield. Since the `TermStructure` machinery already takes care of range-checking, the call to the `Interpolation` instance includes a `true` argument. This causes the value to be extrapolated if the passed time is outside the given range.

Finally, the `InterpolatedZeroCurve` class also defines a few protected constructors. They take the same arguments as the constructors of its parent class `ZeroYieldStructure`, as well as an optional `Interpolator` instance, and forward them to the corresponding parent-class constructors; however, they don't create the interpolation—they cannot, since they don't take any zero-yield data. These constructors are defined so that it is possible to inherit from `InterpolatedZeroCurve`; derived classes will provide the data and create the interpolation based on whatever arguments they take (an example of this will be shown in the remainder of this section). For the same reason, most data members are declared as `mutable`; as noted in the next aside, this makes it possible for derived classes to update the interpolation lazily, should their data change.

---

<sup>6</sup>The `Interpolation` class is described in appendix A, together with a few available interpolations and the corresponding traits.

### Aside: symmetry break.

You might argue that, as in George Orwell's *Animal Farm*, some term structures are more equal than others. The discount-based implementation seems to have a privileged role, being used in the base `YieldTermStructure` class. A more symmetric implementation might define three abstract methods in the base class (`discountImpl`, `zeroYieldImpl`, and `forwardImpl`, to be called from the corresponding public methods) and provide three adapters, adding a `DiscountStructure` class to the existing ones.

Well, the argument is sound; in fact, the very first implementation of the `YieldTermStructure` class was symmetric. The switch to the discount-based interface and the reasons thereof are now lost in the mists of time, but might have to do with the use of `InterestRate` instances; since they can require changes of frequency or compounding, `zeroYield` (to name one method) wouldn't be allowed to return the result of `zeroYieldImpl` directly anyway.

### Aside: twin classes.

You might guess that code for interpolated discount and forward curves would be very similar to that for the interpolated zero-yield curve described here. The question naturally arises: would it be possible to abstract out common code? Or maybe we could even do with a single class template?

The answers are yes and no, respectively. Some code can be abstracted in a template class (in fact, this has been done already). However, the curves must implement three different abstract methods (`discountImpl`, `forwardImpl`, and `zeroYieldImpl`) so we still need all three classes as well as the one containing the common code.

### 3.2.3 Example: bootstrapping an interpolated curve

In this section, we'll build an all-purpose yield-curve template. Building upon the classes described in the previous subsections, we'll give it the ability to interpolate in a number of ways on either discount factors, zero yields, or instantaneous forward rates. The nodes of the curve will be bootstrapped from quoted—and possibly varying—market rates.

Needless to say, this example is a fairly complex one. The class template I'm about to describe, called `PiecewiseYieldCurve`, makes use of a few helper classes, as well as a few template tricks. I'll try and explain all of them as needed.

The implementation of our class template is shown in the following listing.

Implementation of the `PiecewiseYieldCurve` class template.

---

```

template <class Traits, class Interpolator,
    template <class> class Bootstrap = IterativeBootstrap>
class PiecewiseYieldCurve
  : public Traits::template curve<Interpolator>::type,
  public LazyObject {
private:
  typedef typename Traits::template curve<Interpolator>::type
    base_curve;
  typedef PiecewiseYieldCurve<Traits,Interpolator,Bootstrap>
    this_curve;
  typedef Bootstrap<this_curve> bootstrap_type;
  typedef typename Traits::helper helper;
public:
  typedef Traits traits_type;
  typedef Interpolator interpolator_type;
  PiecewiseYieldCurve(
    Natural settlementDays,
    const Calendar& calendar,
    const std::vector<shared_ptr<helper> >& instruments,
    const DayCounter& dayCounter,
    const Interpolator& i = Interpolator(),
    const bootstrap_type& b = bootstrap_type());
  // inspectors not shown

  void update();
private:
  void performCalculations() const;
  DiscountFactor discountImpl(Time) const;
  std::vector<shared_ptr<helper> > instruments_;

  friend class Bootstrap<this_curve>;
  friend class BootstrapError<this_curve>;
  Bootstrap<this_curve> bootstrap_;
};

template <class T, class I,
    template <class> class B>
PiecewiseYieldCurve<T,I,B>::PiecewiseYieldCurve(
  Natural settlementDays,
  const Calendar& calendar,
  const std::vector<shared_ptr<helper> >& instruments,

```

```

        const DayCounter& dayCounter,
        const I& interpolator,
        const B<this_curve>& bootstrap)
: base_curve(settlementDays, calendar, dayCounter, interpolator),
  instruments_(instruments), bootstrap_(bootstrap) {
    bootstrap_.setup(this);
}

template <class T, class I,
          template <class> class B>
void PiecewiseYieldCurve<T,I,B>::update() {
    base_curve::update();
    LazyObject::update();
}

template <class T, class I,
          template <class> class B>
void PiecewiseYieldCurve<T,I,B>::performCalculations() const {
    bootstrap_.calculate();
}

template <class T, class I,
          template <class> class B>
DiscountFactor
PiecewiseYieldCurve<T,I,B>::discountImpl(Time t) const {
    calculate();
    return base_curve::discountImpl(t);
}

```

---

The class takes three template arguments. The first two determine the choice of the underlying data and the interpolation method, respectively; our goal is to instantiate the template as, say,

```
PiecewiseYieldCurve<Discount,LogLinear>
```

or some such combination. I'll refer to the first parameter as the *bootstrap traits*; the second is the interpolator [already described](#). The third, and seemingly ungainly, parameter specifies a class which implements the bootstrapping algorithm. The parameter has a default value (the `IterativeBootstrap` class, which I'll describe later); it is provided so that interested developers can replace the bootstrapping algorithm with another one, either provided by the library or of their own creation. If you're not familiar with its syntax, it's a template template parameter ([Simonis and Weiss, 2001](#)). The repetition is not an error, as suspected by my spell checker as I write these lines;

it means that the template parameter should be an uninstantiated class template (in this case, one taking a single template argument) rather than a typename.<sup>7</sup>

Before declaring the class interface, we need another bit of template programming to determine the parent class of the curve. On the one hand, we inherit our term structure from the `LazyObject` class; as described in chapter 2, this will enable the curve to re-bootstrap itself when needed. On the other hand, we want to inherit it from one of the interpolated curves described in [a previous section](#); depending on the choice of underlying data (discount, zero yields, or forward rates), we must select one of the available class templates and instantiate it with the chosen interpolator class. This is done by storing the class template in the bootstrap traits. Unluckily, C++ didn't allow template typedefs until recently.<sup>8</sup> Therefore, the traits define an inner class template `curve` which takes the interpolator as its template parameter and defines the instantiated parent class as a typedef; this can be seen in the definition of the `Discount` traits, partially shown in the listing below.

Sketch of the `Discount` bootstrap traits.

---

```
struct Discount {
    template <class Interpolator>
    struct curve {
        typedef InterpolatedDiscountCurve<Interpolator> type;
    };
    typedef BootstrapHelper<YieldTermStructure> helper;

    // other static methods
}
```

---

The described machinery allows us to finally refer to the chosen class by using the expression

`Traits::template curve<Interpolator>::type`

that we can add to the list of parent classes.<sup>9</sup>

For an instantiation with `Discount` as bootstrap traits and `LogLinear` as interpolator, the above works out as

`Discount::curve<LogLinear>::type`

which in turn corresponds—as desired—to

---

<sup>7</sup>An uninstantiated template is something like `std::vector`, as opposed to an instantiated one like `std::vector<double>`. The second names a type; the first doesn't.

<sup>8</sup>Template aliases were introduced in the 2011 revision of the C++ standard.

<sup>9</sup>For those unfamiliar with the dark corners of template syntax, the `template` keyword in the expression is a hint for the compiler. When reading this expression, the compiler doesn't know what `Traits` is and has no means to determine that `Traits::curve` is a class template. Adding the keyword gives it the information required for processing the rest of the expression correctly.

```
InterpolatedDiscountCurve<LogLinear>
```

as can be seen from the definition of the `Discount` class.

We make a last short stop before we finally implement the curve; in order to avoid long template expressions, we define a few `typedefs`, namely, `base_curve`, `this_curve`, `bootstrap_type`, and `helper`. The first one refers to the parent class; the second one refers to the very class we're declaring; the third one is the instantiated bootstrap class; and the last one extracts from the bootstrap traits the type of a helper class. This class will be described later in the example; for the time being, I'll just say that it provides the quoted value of an instrument, as well as the means to evaluate such instrument on the term structure being bootstrapped. The aim of the bootstrap will be to modify the curve until the two values coincide. Finally, two other `typedefs` (`traits_type` and `interpolation_type`) store the two corresponding template arguments so that they can be retrieved later.

The actual interface, at last. The constructors (of which only one is shown in the listing) take the arguments required for instantiating the parent interpolated curve, as well as a vector of helpers and possibly the interpolation or bootstrap algorithms. Next, a number of inspectors (such as `times` or `data`) are defined, overriding the versions in the parent class; as we'll see, this allows them to ensure that the curve is fully built before returning the required data. The public interface is completed by the `update` method.

The protected methods include `performCalculations`, needed to implement the `LazyObject` interface; and `discountImpl`, which (like the public inspectors) overrides the parent-class version. The `Bootstrap` class template is instantiated with the type of the curve being defined. Since it will need access to the internals of the curve, the resulting class is declared as a friend of the `PiecewiseYieldCurve` class; the same is done for the `BootstrapError` class, used in the bootstrap algorithm and described later. Finally, we store the helpers and the passed instance of the bootstrap class.

Let's now have a look at the implementation. The constructor holds no surprises: it passes the needed arguments to the base class and stores in this class the other ones. Finally, it passes the curve itself to the stored `Bootstrap` instance and makes it perform some preliminary work—more on this later. The `update` method is needed for disambiguation; we are inheriting from two classes (`LazyObject` and `TermStructure`) which both define their implementation of the method. The compiler justly refuses to second-guess us, so we have to explicitly call both parent implementations. As for the `performCalculations` method, it delegates its work to the `Bootstrap` instance.

Lastly, a look at the `discountImpl` method shows us why such method had to be overridden; before calling the parent-class implementation, it has to ensure that the data were bootstrapped by calling the `calculate` method. This also holds for the other overridden inspectors, all following the same pattern. Instead, there's no need to override the `zeroYieldImpl` and `forwardImpl` methods, even if the curve inherits from `ZeroYieldStructure` or `ForwardRateStructure`; if you look back at the listing of the `ZeroYieldStructure` class, you'll see that those methods are only ever called by `discountImpl`. Thus, overriding the latter is enough.

At this point, I need to describe the `BootstrapHelper` class template; its interface is sketched in the next listing.

Interface of the `BootstrapHelper` class template.

---

```
template <class TS>
class BootstrapHelper : public Observer, public Observable {
public:
    BootstrapHelper(const Handle<Quote>& quote);
    virtual ~BootstrapHelper() {}

    Real quoteError() const;
    const Handle<Quote>& quote() const;
    virtual Real impliedQuote() const = 0;

    virtual void setTermStructure(TS*);

    virtual Date latestDate() const;

    virtual void update();
protected:
    Handle<Quote> quote_;
    TS* termStructure_;
    Date latestDate_;
};
```

---

For our curve, we'll instantiate it (as you can see in the `Discount` traits shown earlier) as `BootstrapHelper<YieldTermStructure>`; for convenience, the library provides an alias to this class called `RateHelper` that can be used in place of the more verbose type name.

Each instance of this class—or rather, of derived classes; the class itself is an abstract one—will help bootstrapping a single node on the curve. The input datum for the node is the quoted value of an instrument; this is provided as a `Handle` to a `Quote` instance, since the value will change in time. For a yield curve, such instruments might be deposits or swaps, quoted as the corresponding market rates. For each kind of instrument, a derived class must be provided.

The functionality that is common to all helpers is implemented in the base class. `BootstrapHelper` inherit from both `Observer` and `Observable`; the double role allows it to register with the market value and notify changes to the curve, signaling the need to perform a new bootstrap. Its constructor takes a `Handle<Quote>` providing the input market value, stores it as a data member, and registers itself as an observer. Three methods deal with the underlying instrument value. The `quote` method returns the handle containing the quoted value; the abstract `impliedQuote` method returns the value as calculated on the curve being bootstrapped; and the convenience method `quoteError` returns the signed difference between the two values.

Two more methods are used for setting up the bootstrap algorithm. The `setTermStructure` method links the helper with the curve being built. The `latestDate` method returns the latest date for which

curve data are required in order to calculate the implied value of the market datum;<sup>10</sup> such date will be used as the coordinate of the node being bootstrapped. The last method, `update`, forwards notifications from the quote to the observers of the helper.

The library provides a few concrete helper classes inherited from `RateHelper`. In the interest of brevity, allow me to do a little hand-waving here instead of showing you the actual code. Each of the helper classes implements the `impliedValue` for a specific instrument and includes code for returning the proper latest date. For instance, the `DepositRateHelper` class forecasts a quoted deposit rate by asking the curve being bootstrapped for the forward rate between its start and maturity dates; whereas the `SwapRateHelper` class forecasts a swap rate by instantiating a `Swap` object, pricing it on the curve, and implying its fair rate. In a multi-curve setting, it's also possible to use the bootstrapped curve for forecasting forward rates and an external curve for discounting. If you're interested, more details on this (and a discussion on what helpers to choose for a given curve) are available in [Ametrano and Bianchetti, 2013](#).

We can finally dive into the bootstrap code. The following listing shows the interface of the `IterativeBootstrap` class, which is provided by the library and used by default.

Sketch of the `IterativeBootstrap` class template.

---

```
template <class Curve>
class IterativeBootstrap {
    typedef typename Curve::traits_type Traits;
    typedef typename Curve::interpolator_type Interpolator;
public:
    IterativeBootstrap(Real accuracy = ...,
                      Real minValue = ...,
                      Real maxValue = ...);
    void setup(Curve* ts);
    void calculate() const;
private:
    Curve* ts_;
    Real accuracy_;
    Real minValue_, maxValue_;
};

template <class Curve>
void IterativeBootstrap<Curve>::calculate() const {
    Size n = ts_->instruments_.size();

    // sort rate helpers by maturity
    // check that no two instruments have the same maturity
    // check that no instrument has an invalid quote
```

---

<sup>10</sup>The latest required date does not necessarily correspond to the maturity of the instrument. For instance, if the instrument were a constant-maturity swap, the curve should extend a few years beyond the swap maturity in order to forecast the rate paid by the last coupon.

```

for (Size i=0; i<n; ++i)
    ts_->instruments_[i]->setTermStructure(
        const_cast<Curve*>(ts_));

ts_->dates_ = std::vector<Date>(n+1);
// same for the other data vectors

ts_->dates_[0] = Traits::initialDate(ts_);
ts_->times_[0] = ts_->timeFromReference(ts_->dates_[0]);
ts_->data_[0] = Traits::initialValue(ts_);

for (Size i=0; i<n; ++i) {
    ts_->dates_[i+1] = ts_->instruments_[i]->latestDate();
    ts_->times_[i+1] =
        ts_->timeFromReference(ts_->dates_[i+1]);
}

Brent solver;

for (Size iteration = 0; ; ++iteration) {
    for (Size i=1; i<n+1; ++i) {
        if (iteration == 0) {
            // extend interpolation a point at a time
            ts_->interpolation_ =
                ts_->interpolator_.interpolate(
                    ts_->times_.begin(),
                    ts_->times_.begin()+i+1,
                    ts_->data_.begin());
            ts_->interpolation_.update();
        }
    }

    Rate guess;
    // estimate guess by using the value at the previous iteration,
    // by extrapolating, or by asking the traits

    // bracket the solution
    Real min = minValue_ != Null<Real>() ? minValue_ :
        Traits::minValueAfter(i, ts_->data_);

    Real max = maxValue_ != Null<Real>() ? maxValue_ :
        Traits::maxValueAfter(i, ts_->data_);
}

```

```

        BootstrapError<Curve> error(ts_, instrument, i);
        ts_->data_[i] = solver.solve(error, accuracy_,
                                      guess, min, max);
    }

    if (!Interpolator::global)
        break;      // no need for convergence loop

    // check convergence and break if tolerance is reached
    // bail out if tolerance wasn't reached in the given number of iterations
}
}

```

---

For convenience, typedefs are defined to extract from the curve the traits and interpolator types. The constructor and the `setup` method are not of particular interest. The first initializes the contained term-structure pointer to a null one and stores a few optional parameters; the second stores the passed curve pointer, checks that we have enough helpers to bootstrap the curve, and registers the curve as an observer of each helper. The bootstrap algorithm is implemented by the `calculate` method. In the version shown here, I'll gloss over a whole lot of details and corner cases; if you're interested, you can peruse the full code in the library.

First, all helpers are set the current term structure. This is done each time (rather than in the `setup` method) to allow a set of helpers to be used with different curves.<sup>11</sup> Then, the data vectors are initialized. The date and value for the initial node are provided by the passed traits; for yield term structures, the initial date corresponds to the reference date of the curve. The initial value depends on the choice of the underlying data; it is 1 for discount factors and a dummy value (which will be overwritten during the bootstrap procedure) for zero or forward rates. The dates for the other nodes are the latest needed dates of the corresponding helpers; the times are obtained by using the available curve facilities.

At this point, we can instantiate the one-dimensional solver that we'll use at each node (more details on this in appendix A) and start the actual bootstrap. The calculation is written as two nested loops; an inner one—the bootstrap proper—that walks over each node, and an outer one that repeats the process. Iterative bootstrap is needed when a non-local interpolation (such as cubic splines) is used. In this case, setting the value of a node modifies the whole curve, invalidating previous nodes; therefore, we must go over the nodes a number of times until convergence is reached.

As I mentioned, the inner loop walks over each node—starting, of course, from the one at the earliest date. During the first iteration (when `iteration == 0`) the interpolation is extended a point at a time; later iterations use the full data range, so that the previous results are used as a starting point and refined. After each node is added, a one-dimensional root-finding algorithm is used to reproduce the corresponding market quote. For the first iteration, a guess for the solution can be provided by

<sup>11</sup>Of course, the same helpers could not be passed safely to different curves in a multi-threaded environment since they would compete for it. Then again, much of QuantLib is not thread-safe, so it's kind of a moot point.

the bootstrap traits or by extrapolating the curve built so far; for further iteration, the previous result is used as the guess. The maximum and minimum value are usually provided by the traits (possibly based on the nodes already bootstrapped in case some kind of monotonicity is required; this used to apply to discount factors, way back when we used to assume rates to be positive) but in extreme cases they can be overridden by providing them directly to the constructor of the instance.

The only missing ingredient for the root-finding algorithm is the function whose zero must be found. It is provided by the `BootstrapError` class template (shown in the next listing), that adapts the helper's `quoteError` calculation to a function-object interface. Its constructor takes the curve being built, the helper for the current node, and the node index, and stores them. Its `operator()` makes instances of this class usable as functions; it takes a guess for the node value, modifies the curve data accordingly, and returns the quote error.

Interface of the `BootstrapError` class template.

---

```
template <class Curve>
class BootstrapError {
    typedef typename Curve::traits_type Traits;
public:
    BootstrapError(
        const Curve* curve,
        const shared_ptr<typename Traits::helper>& helper,
        Size segment);
    Real operator()(Rate guess) const {
        Traits::updateGuess(curve_->data_, guess, segment_);
        curve_->interpolation_.update();
        return helper_->quoteError();
    }
private:
    const Curve* curve_;
    const shared_ptr<typename Traits::helper> helper_;
    const Size segment_;
};
```

---

At this point, we're all set. The inner bootstrap loop creates a `BootstrapError` instance and passes it to the solver, which returns the node value for which the error is zero—i.e., for which the implied quote equals (within accuracy) the market quote. The curve data are then updated to include the returned value, and the loop turns to the next node.

When all nodes are bootstrapped, the outer loop checks whether another iteration is necessary. For local interpolations, this is not the case and we can break out of the loop. For non-local ones, the obtained accuracy is checked (I'll spare you the details here) and iterations are added until convergence is reached.

This concludes the bootstrap; and, as I don't want to further test your patience, it also concludes

the example. Sample code using the `PiecewiseYieldCurve` class can be found in the QuantLib distribution, e.g., in the swap-valuation example.

### Aside: a friend in need.

“Wait a minute,” you might have said upon looking at the `PiecewiseYieldCurve` declaration, “wasn’t `friend` considered harmful?” Well, yes—`friend` declarations break encapsulation and force tight coupling of classes.

However, let’s look at the alternatives. The `Bootstrap` class needs write access to the curve data. Beside declaring it as a friend, we might have given it access in three ways. On the one hand, we might have passed the data to the `Bootstrap` instance; but this would have coupled the two classes just as tightly (the curve internals couldn’t be changed without changing the bootstrap code as well). On the other hand, we might have exposed the curve data through its public interface; but this would have been an even greater break of encapsulation (remember that we need write access). And on the gripping hand, we might encapsulate the data in a separate class and use private inheritance to control access. At the time of release 1.0, we felt that the `friend` declaration was no worse than the first two alternatives and resulted in simpler code. The third (and best) alternative might be implemented in a future release.

### 3.2.4 Example: adding z-spread to an interest-rate curve

This example (a lot simpler than the previous one) shows how to build a term-structure based on another one. We’ll take an existing risk-free curve and modify it to include credit risk. The risk is expressed as a z-spread, i.e., a constant spread to be added to the zero-yield rates. For the pattern-savvy, this is an application of the Decorator pattern; we’ll wrap an existing object, adding some behavior and delegating the rest to the original instance.

The implementation of the `ZeroSpreadedTermStructure` is shown in the next listing.

Implementation of the `ZeroSpreadedTermStructure` class.

---

```
class ZeroSpreadedTermStructure : public ZeroYieldStructure {
public:
    ZeroSpreadedTermStructure(
        const Handle<YieldTermStructure>& h,
        const Handle<Quote>& spread);
    : originalCurve_(h), spread_(spread) {
        registerWith(originalCurve_);
        registerWith(spread_);
    }
    const Date& referenceDate() const {
        return originalCurve_->referenceDate();
```

```

    }
    DayCounter dayCounter() const {
        return originalCurve_->dayCounter();
    }
    Calendar calendar() const {
        return originalCurve_->calendar();
    }
    Natural settlementDays() const {
        return originalCurve_->settlementDays();
    }
    Date maxDate() const {
        return originalCurve_->maxDate();
    }
protected:
    Rate zeroYieldImpl(Time t) const {
        InterestRate zeroRate =
            originalCurve_->zeroRate(t, Continuous,
                                         NoFrequency, true);
        return zeroRate + spread_->value();
    }
private:
    Handle<YieldTermStructure> originalCurve_;
    Handle<Quote> spread_;
};

```

---

As previously mentioned, it is based on zero-yield rates; thus, it inherits from the `ZeroYieldStructure` adapter described in [a previous section](#) and will have to implement the required `zeroYieldImpl` method. Not surprisingly, its constructor takes as arguments the risk-free curve to be modified and the z-spread to be applied; to allow switching data sources, both are passed as handles. The arguments are stored in the corresponding data members, and the curve registers with both of them as an observer. The `update` method inherited from the base class will take care of forwarding any received notifications.

Note that none of the base-class constructors was called explicitly. As you might remember from [a previous section](#), this means that instances of our curve store no data that can be used by the `TermStructure` machinery; therefore, the class must provide its own implementation of the methods related to reference-date calculation. In true Decorator fashion, this is done by delegating behavior to the wrapped object; each of the `referenceDate`, `dayCounter`, `calendar`, `settlementDays`, and `maxDate` methods forwards to the corresponding method in the risk-free curve.

Finally, we can implement our own specific behavior—namely, adding the z-spread. This is done in the `zeroYieldImpl` method; we ask the risk-free curve for the zero-yield rate at the required

time (continuously compounded, since that's what our method must return), add the value of the z-spread, and return the result as the new zero-yield rate. The machinery of the `ZeroYieldStructure` adapter will take care of the rest, giving us the desired risky curve.

## 3.3 Other term structures

So far, the focus of this chapter has been on yield term structures. Of course, other kinds of term structure are implemented in the library. In this section, I'll review them shortly: mostly, I'll point out how they differ from yield term structures and what particular features they sport.

### 3.3.1 Default-probability term structures

Default-probability term structures are the most similar in design to yield term structures. They can be expressed in terms of default probability, survival probability, default density, or hazard rate; any one of the four quantities can be obtained from any other, much like zero rates and discount factors.

Unlike yield term structures (in which all methods are implemented in terms of the `discountImpl` method) the base default-probability structure has no single method for others to build upon. Instead, as shown in the listing below, it declares two abstract methods `survivalProbabilityImpl` and `defaultDensityImpl`. It's left to derived classes to decide which one should be written in terms of the other; the base class implements `survivalProbability` and `defaultDensity` based on the respective implementation methods,<sup>12</sup> `defaultProbability` based trivially on `survivalProbability`, and `hazardRate` in terms of both survival probability and default density.

Sketch of the `DefaultProbabilityTermStructure` class.

---

```
class DefaultProbabilityTermStructure : public TermStructure {
public:
    ...constructors...

    Probability survivalProbability(Time t) const {
        return survivalProbabilityImpl(t);
    }

    Probability defaultProbability(Time t) const {
        return 1.0 - survivalProbability(t);
    }

    Probability defaultProbability(Time t1,
                                   Time t2) const {
        Probability p1 = defaultProbability(t1),
        p2 = defaultProbability(t2);
```

<sup>12</sup>The implementation of `survivalProbability` and `defaultDensity` is not as simple as shown, of course; here I omitted range checking and extrapolation for clarity.

```

        return p2 - p1;
    }

    Real defaultDensity(Time t) const {
        return defaultDensityImpl(t);
    }

    Rate hazardRate(Time t) const {
        Probability S = survivalProbability(t);
        return S == 0.0 ? 0.0 : defaultDensity(t)/S;
    }

    ...other methods...
protected:
    virtual Probability survivalProbabilityImpl(Time) const = 0;
    virtual Real defaultDensityImpl(Time) const = 0;
private:
    ...data members...
};
```

---

The next listing sketches the adapter classes that, as for yield term structures, allow one to define a new default-probability structure in terms of the single quantity of his choice—either survival probability, default density, or hazard rate (the default probability is so closely related to survival probability that we didn’t think it necessary to provide a corresponding adapter).

Adapter classes for default-probability term structures.

---

```

class SurvivalProbabilityStructure
    : public DefaultProbabilityTermStructure {
public:
    ...constructors...
protected:
    Real defaultDensityImpl(Time t) const {
        // returns the (numerical) derivative
        // of the survival probability at t
    }
};

class DefaultDensityStructure
    : public DefaultProbabilityTermStructure {
public:
    ...constructors...
protected:
```

```

        Probability survivalProbabilityImpl(Time t) const {
            // returns 1 minus the integral of the default density from 0 to t
        }
    };

class HazardRateStructure
    : public DefaultProbabilityTermStructure {
public:
    ...constructors...
protected:
    virtual Real hazardRateImpl(Time) const = 0;
    Probability survivalProbabilityImpl(Time t) const {
        // returns exp(-I), where I is the integral
        // of the hazard rate from 0 to t
    }
    Real defaultDensityImpl(Time t) const {
        return hazardRateImpl(t)*survivalProbabilityImpl(t);
    }
}

```

---

The first one, `SurvivalProbabilityStructure`, defines `defaultDensityImpl` in terms of the implementation of `survivalProbabilityImpl`, which is left purely virtual and must be provided in derived classes; the second one, `DefaultDensityStructure`, does the opposite; and the last one, `HazardRateStructure`, defines both survival probability and default density in terms of a newly-declared purely abstract `hazardRateImpl` method.

Unfortunately, some of the adapters rely on numerical integration in order to provide conversions among the desired quantities. The provided implementations use dark magic in both maths and coding (namely, Gaussian quadratures and `boost::bind`) to perform the integrations efficiently; but when inheriting from such classes, you should consider overriding the adapter methods if a closed formula is available for the integral.

Like for yield curves, the library provides a few template classes that implement the adapter interfaces by interpolating discrete data, as well as a generic piecewise default-probability curve template and the traits required to select its underlying quantity. Together with the existing interpolation traits, this allows one to instantiate classes such as `PiecewiseDefaultCurve<DefaultDensity, Linear>`. The implementation is quite similar to the one described for the `PiecewiseYieldCurve` class template; in fact, so much similar that it's not worth describing here. The only thing worth noting is that the default-probability structure is not self-sufficient: in order to price the instruments required for its bootstrap, a discount curve is needed (then again, the same is true of LIBOR curves in today's multiple-curve settings). You'll have to be consistent and use the same curve for your pricing engines; otherwise, you might suddenly find out that your CDS are no longer at the money.

## Aside: Cinderella method.

In the implementation of `DefaultProbabilityTermStructure`, you've probably noticed another symmetry break like the one discussed in [an earlier aside](#). There's a difference though; in that case, discount factors were singled out to be given a privileged role. In this case, hazard rates are singled out to play the mistreated stepsister; there's no `hazardRateImpl` beside the similar methods declared for survival probability or default density. Again, a look at past versions of the code shows that once, it was symmetric; and again, I can give no reason for the change. I'm sure it looked like a good idea at the time.

The effect is that classes deriving from the `HazardRateStructure` adapter must go through some hoops to return hazard rates, since they're not able to call `hazardRateImpl` directly; instead, they have to use the default implementation and return the ratio of default density and survival probability (possibly performing an integration along the way). Unfortunately, even our fairy godmother can't change this now without risking to break existing code.

### 3.3.2 Inflation term structures

Inflation term structures have a number of features that set them apart from the term structures I described so far. Not surprisingly, most such features add complexity to the provided classes.

The most noticeable difference is that we have two separate kinds of inflation term structures (and two different interfaces) instead of a single one. The library does provide a single base class `InflationTermStructure`, that contains some inspectors and some common behavior; however, the interfaces returning the actual inflation rates are declared in two separate child classes, leading to the hierarchy sketched in the listing below. The two subclasses model zero-coupon and year-on-year inflation rates, which are not easily converted into one another and thus foil our usual multiple-adapter scheme.

Sketch of the `InflationTermStructure` class and its children.

---

```
class InflationTermStructure : public TermStructure {
public:
    ...constructors...

    virtual Date baseDate() const = 0;
    virtual Rate baseRate() const;
    virtual Period observationLag() const;

    Handle<YieldTermStructure> nominalTermStructure() const;

    void setSeasonality(const shared_ptr<Seasonality>&);

protected:
```

```

        Handle<YieldTermStructure> nominalTermStructure_;
        Period observationLag_;
        ...other data members...
    };

    class ZeroInflationTermStructure
        : public InflationTermStructure {
    public:
        ...constructors...
        Rate zeroRate(const Date &d,
                      const Period& instObsLag = Period(-1,Days),
                      bool forceLinearInterpolation = false,
                      bool extrapolate = false) const;
    protected:
        virtual Rate zeroRateImpl(Time t) const = 0;
    };

    class YoYInflationTermStructure
        : public InflationTermStructure {
    public:
        ...constructors...
        Rate yoyRate(const Date &d,
                      const Period& instObsLag = Period(-1,Days),
                      bool forceLinearInterpolation = false,
                      bool extrapolate = false) const;
    protected:
        virtual Rate yoyRateImpl(Time time) const = 0;
    };

```

---

This state of things has both advantages and disadvantages; possibly more of the latter. On the one hand, it leads to a pair of duplicated sub-hierarchies, which is obviously a smell.<sup>13</sup> On the other hand, it simplifies a bit the sub-hierarchies; for instance, there's no adapter classes since each kind of term structure has only one underlying quantity (that is, either zero-coupon rates or year-on-year rates).

Other differences are due to the specific quirks of inflation fixings. Since inflation figures for a given month are announced after an observation lag, inflation term structures have a base date, as well as a reference date; the base date is the one corresponding to the latest announced fixing. If an inflation figure is needed for a date in the past with respect to today's date but after the base date, it must be forecast.<sup>14</sup> Also, since inflation fixings are affected by seasonality, inflation term structures provide the means to store an instance of the polymorphic `Seasonality` class (which for brevity I won't

<sup>13</sup>It can get worse. Until now, we haven't considered period-on-period rates with a frequency other than annual. Hopefully, they will only lead to a generalization of the year-on-year curve.

<sup>14</sup>You might remember a footnote at the beginning of this chapter where I obscurely suggested that the future doesn't always begin at the reference date. This is the exception I was referring to.



```

        BusinessDayConvention bdc,
const DayCounter& dc = DayCounter());

virtual BusinessDayConvention businessDayConvention() const {
    return bdc_;
}
Date optionDateFromTenor(const Period&) const {
    return calendar().advance(referenceDate(),
                           p,
                           businessDayConvention());
}

virtual Rate minStrike() const = 0;
virtual Rate maxStrike() const = 0;
protected:
    void checkStrike(Rate strike, bool extrapolate) const;
private:
    BusinessDayConvention bdc_;
};
```

---

The class adds two things to `TermStructure`, from which it inherits. The first is a method, `optionDateFromTenor`, that calculates the exercise date of an option from its tenor; to do this, it used the calendar provided by the base-class interface as well as a business-day convention stored in this class (which is passed to the constructors, and from which the usual inspector is provided). Instead, this functionality could be encapsulated in some kind of utility class and used elsewhere.<sup>15</sup>

The second addition involves two pure virtual methods that return the minimum and maximum strike over which the term structure is defined and a protected method that checks a given strike against the defined range. Unfortunately, these don't make sense for a local volatility structure; but leaving them out would require yet another level of hierarchy to hold them (namely, an implied-vol structure class) so I'm not shocked to see them here instead.

### 3.3.4 Equity volatility structures

Equity and FX-rate Black volatilities are modeled by the `BlackVolTermStructure` class, shown in the following listing.

---

<sup>15</sup>The idea of such date calculator was suggested years ago on the mailing list by someone who, hopefully, will forgive me if I can no longer recall nor find out their name.

Partial interface of the `BlackVolTermStructure` class.

---

```

class BlackVolTermStructure : public VolatilityTermStructure {
public:
    Volatility blackVol(const Date& maturity,
                        Real strike,
                        bool extrapolate = false) const;
    Volatility blackVol(Time maturity,
                        Real strike,
                        bool extrapolate = false) const;
    Real blackVariance(const Date& maturity,
                       Real strike,
                       bool extrapolate = false) const;
    Real blackVariance(Time maturity,
                       Real strike,
                       bool extrapolate = false) const;
    Volatility blackForwardVol(const Date& date1,
                               const Date& date2,
                               Real strike,
                               bool extrapolate = false) const;
    Real blackForwardVariance(const Date& date1,
                             const Date& date2,
                             Real strike,
                             bool extrapolate = false) const;
    // same two methods as above, taking two times
protected:
    virtual Real blackVarianceImpl(Time t,
                                    Real strike) const = 0;
    virtual Volatility blackVolImpl(Time t,
                                    Real strike) const = 0;
};
```

---

Apart from its several constructors (which, as usual, forward their arguments to the base class and would be made unnecessary by constructor inheritance, introduced in C++11) the class defines the overloaded `blackVol` method to retrieve the volatility  $\sigma$  for a given exercise date or time; the `blackVariance` method for the corresponding variance  $\sigma^2 t$ ; and the `blackForwardVol` and `blackForwardVariance` methods for the forward volatility and variance between two future dates.

Following the Template Method pattern (no surprise there) all these methods are implemented by calling the protected and pure virtual `blackVolImpl` and `blackVarianceImpl` methods. The public interface adds range checking and, in the case of forward volatility or variance, the bit of logic required to calculate the forward values from the spot values at the two passed dates.

At the time of this writing, the `BlackVolTermStructure` class also defines a private `static const` data member `dT` that is no longer used—much like we still carry around an appendix, or a vestigial

tailbone. By the time you read this, I hope to have it removed. The data member, I mean. Not my appendix.

As usual, adapters are provided to write only one of the `blackVolImpl` or the `blackVarianceImpl` method; they are shown in the next listing.<sup>16</sup> It is unfortunate that the name of one of them, the `BlackVolatilityTermStructure` class, is so confusingly similar to the name of the base class. I'm open to suggestions for changing either one in a future version of the library.

Adapters for the `BlackVolTermStructure` class.

---

```

class BlackVolatilityTermStructure
    : public BlackVolTermStructure {
    ... // constructors, not shown
protected:
    Real blackVarianceImpl(Time maturity, Real strike) const {
        Volatility vol = blackVolImpl(t, strike);
        return vol*vol*t;
    }
};

class BlackVarianceTermStructure
    : public BlackVolTermStructure {
    ... // constructors, not shown
protected:
    Volatility blackVolImpl(Time t, Real strike) const {
        Time nonZeroMaturity = (t==0.0 ? 0.00001 : t);
        Real var = blackVarianceImpl(nonZeroMaturity, strike);
        return std::sqrt(var/nonZeroMaturity);
    }
};

```

---

## Aside: interpolations and extrapolations.

One of the available volatility classes is the `BlackVarianceSurface` class, which interpolates a matrix of quoted Black volatilities. I won't describe it here, since you're probably sick of term-structure examples by now; but it has a couple of interesting features.

The first is that the interpolation can be changed once the structure is built; the relevant method is

```

template <class Interpolator>
void setInterpolation(const Interpolator& i) {
    varianceSurface_ =
        i.interpolate(times_.begin(), times_.end(),

```

---

<sup>16</sup>For simple examples of either kind, you can look at the `ConstantBlackVol` and the misleadingly-named `ImpliedVolTermStructure` classes in the library.

```

        strikes_.begin(), strikes_.end(),
        variances_);
    notifyObservers();
}

```

This is not possible in other interpolated curves, in which the type of the interpolation is a template argument and is fixed at instantiation; see, for instance, the `PiecewiseYieldCurve` class template. The difference is that `BlackVarianceSurface` doesn't need to store the interpolator, and thus doesn't need to know its type outside the `setInterpolation` method.

The second feature of `BlackVarianceSurface` is the possibility to customize the kind of extrapolation to use when the passed strike is outside the range of the interpolation. It is possible either to extend the underlying interpolation or to extrapolate flatly the value at the end of the range; the behavior at either end can be specified independently.

Now, it would be nice if this behavior could be extracted in some base class and reused. The choice of extrapolation can be implemented in a generic way; given any interpolation  $f$  defined up to  $x_{\max}$  (or down to  $x_{\min}$ ), and given an  $x > x_{\max}$ , the two choices can be realized by returning  $f(x)$  for extension of  $f(x_{\max})$  for flat extrapolation.

The `Extrapolator` class would seem the obvious choice for defining such behavior; but, unfortunately, this wouldn't work if we still want to make different choices on different boundaries. As a base class, `Extrapolator` would have no knowledge of the fact that, for instance, an interest-rate structure is defined over a time range whose lower bound is 0, while a volatility surface also has a range of strikes. Since it can't distinguish between boundaries, `Extrapolator` can't define an interface that specifies behavior on any of them; we'd be forced to make a single choice and apply it everywhere.

Therefore, the only possibility I see for code reuse at this time would be to define a polymorphic `Extrapolation` class, code the different behaviors into derived classes, and store the required number of instances into any given term structure.

### 3.3.5 Interest-rate volatility structures

Last in our tour are interest-rate volatilities. There are three different hierarchies of them, each with its quirks.

The first hierarchy models cap and floor term volatilities; the base class is the `CapFloorTermVolatilityStructure` shown in the next listing.

Interface of the `CapFloorTermVolatilityStructure` class.

---

```
class CapFloorTermVolatilityStructure
    : public VolatilityTermStructure {
public:
    ... // constructors, not shown
    Volatility volatility(const Period& length, Rate strike,
                          bool extrapolate = false) const;
    Volatility volatility(const Date& end, Rate strike,
                          bool extrapolate = false) const;
    Volatility volatility(Time t, Rate strike,
                          bool extrapolate = false) const;
protected:
    virtual Volatility volatilityImpl(Time length,
                                       Rate strike) const = 0;
};
```

---

It is a straightforward application of the patterns seen so far, with one main difference; the volatility is not dependent on the exercise time, which is fixed to today's time, but on the maturity of the strip of caplets or floorlets of which the instrument is composed.

The difference is semantic: `volatility(t,strike)` has for this class a different meaning, even though the interface and the implementation are the same as for the other volatility classes. In turn, this has a couple of small consequences on the interface: on the one hand, there's an additional overload of the `volatility` method that takes the length of the cap as a `Period` instance, as seemed natural; and on the other hand, the volatility doesn't really measure the width of the distribution of any variable at time  $T$ , so the `variance` method was omitted.

The second hierarchy models the volatilities of single caplets and floorlets;<sup>17</sup> its base class is the `OptionletVolatilityStructure` class, shown in the listing below.

Interface of the `OptionletVolatilityStructure` class.

---

```
class OptionletVolatilityStructure
    : public VolatilityTermStructure {
public:
    ... // constructors, not shown
    Volatility volatility(const Period& optionTenor,
                          Rate strike,
                          bool extrapolate = false) const;
    Volatility volatility(const Date& optionDate,
                          Rate strike,
                          bool extrapolate = false) const;
```

---

<sup>17</sup>There are ways to convert from cap to caplet volatilities, of course, but I won't cover them here. Look into QuantLib for `OptionletStripper` and its derived classes.

```

Volatility volatility(Time optionTime,
                     Rate strike,
                     bool extrapolate = false) const;

Real blackVariance(const Period& optionTenor,
                    Rate strike,
                    bool extrapolate = false) const;
// same overloads as for `volatility`


shared_ptr<SmileSection> smileSection(
    const Period& optionTenor,
    bool extrapolate = false) const;
shared_ptr<SmileSection> smileSection(
    const Date& optionDate,
    bool extrapolate = false) const;
shared_ptr<SmileSection> smileSection(
    Time optionTime,
    bool extrapolate = false) const;

protected:
    virtual shared_ptr<SmileSection> smileSectionImpl(
        const Date& optionDate) const;
    virtual shared_ptr<SmileSection> smileSectionImpl(
        Time optionTime) const = 0;

    virtual Volatility volatilityImpl(const Date& d,
                                      Rate strike) const {
        return volatilityImpl(timeFromReference(d), strike);
    }
    virtual Volatility volatilityImpl(Time optionTime,
                                      Rate strike) const = 0;
};
```

---

The semantics are back to normal, with the volatility being dependent on the exercise time. The structure of the class is as usual, too, but with a notable addition: besides the usual `volatility` method, the class declares a `smileSection` method that takes an exercise date (or the corresponding time or period) and returns an object that models the whole smile at that date and inherits from the base class `SmileSection`, shown in the next listing. The interface is modeled after that of the `volatility` class and shouldn't need explanation, apart from noting that the time is fixed and doesn't need to be passed as an argument to the various methods.

Partial interface of the `SmileSection` class.

---

```
class SmileSection : public virtual Observable,
                     public virtual Observer {

public:
    SmileSection(Time exerciseTime,
                 const DayCounter& dc = DayCounter());
    virtual ~SmileSection() {}

    virtual Real minStrike() const = 0;
    virtual Real maxStrike() const = 0;
    Real variance(Rate strike) const;
    Volatility volatility(Rate strike) const;
    virtual Real atmLevel() const = 0;

protected:
    virtual Real varianceImpl(Rate strike) const;
    virtual Volatility volatilityImpl(Rate strike) const = 0;
};
```

---

Simple as it seems, the addition of smile sections yields a new design that sees volatilities not as surfaces, but as collections of smiles at different times. This opens up the possibility to model the smile directly and to reuse the corresponding classes across different types of volatilities (say, for both cap/floors and swaptions).

However, we weren't bold enough to switch completely to the new interface. The two representations (the surface and the series of smiles) still coexist in the base caplet-volatility class, leading to some disadvantages. Any derived class can implement `volatilityImpl` in terms of `smileSectionImpl` as

```
Volatility volatilityImpl(Time t, Real strike) const {
    return smileSectionImpl(t).volatility(strike);
}
```

but this is only convenient if we're modeling the smile to begin with. If we're modeling the volatility surface, instead, this design makes it a lot more cumbersome to implement a new class. There's no easy way to implement `smileSectionImpl` in terms of `volatilityImpl`: we should return an object that's able to call `volatilityImpl` from its own `volatility` method, but that would link the lifetimes of the smile section and the volatility surface, and in turn raise all kind of problems. We would probably end up writing a smile-section class that contains the same code as the volatility surface; therefore, we might as well drop `volatilityImpl` altogether.

Unfortunately, naive implementations of the `smileSectionImpl` method cause new objects to be allocated at each call, which is obviously not good for performance. Smarter implementations would need to cache objects, and with this comes more complexity. Thus, the smile section is an interesting

concept, but maybe more trouble than it's worth. It might be reduced in scope, and used as an implementation detail for classes that model the smile directly.

A final note on the `OptionletVolatilityStructure` class: unlike the classes we've seen so far, it also declares an overload of `volatilityImpl` that takes a date. It has a default implementation that converts the date to a time and calls the other overload, so writers of derived classes don't need to override it; but it can increase accuracy when the derived class takes and stores dates as input.

Finally, the third hierarchy models swaption volatilities; its base class is `SwaptionVolatilityStructure`, shown in the listing below.

Interface of the `SwaptionVolatilityStructure` class.

---

```
class SwaptionVolatilityStructure
    : public VolatilityTermStructure {

public:
    ... // constructors, not shown
    Volatility volatility(const Period& optionTenor,
                          const Period& swapTenor,
                          Rate strike,
                          bool extrapolate = false) const;
    // various overloads, also for `blackVariance` and `smileSection`

    virtual const Period& maxSwapTenor() const = 0;
    Time maxSwapLength() const;

protected:
    virtual Volatility volatilityImpl(Time optionTime,
                                       Time swapLength,
                                       Rate strike) const = 0;
    void checkSwapTenor(Time swapLength,
                        bool extrapolate) const;
};
```

---

It has the same new features I just described for caplet volatilities (the use of smile sections and the overload of `volatilityImpl`) and a new shtick: the additional dimension given by the length of the underlying swap, which in turn brings a few more methods for specifying and checking the range of the corresponding argument.

\* \* \*

That's all for term structures. You'll find plenty of examples in the library; I hope this chapter will help you make sense of them.

# 4. Cash flows and coupons

“Cash is king,” says a sign in the office of StatPro’s CFO.<sup>1</sup> In order to deal with the comings and goings of its Majesty (yes, I’m ironic), QuantLib must provide the means not only to price, but also to analyze coupon-bearing instruments such as bonds and interest-rate swaps. This chapter describes the classes that model different kinds of cash flows and coupons.

## 4.1 The `CashFlow` class

As usual, we start at the top of the class hierarchy. The `CashFlow` class provides the basic interface for all cash flows. As this level of abstraction, the information is of course rather poor; namely, the only provided inspectors return the date and amount of the cash flow. To save one from comparing dates explicitly (and for consistency, as explained in [a later aside](#)) another convenience method tells whether the cash flow has already occurred at a given date. Finally, yet another method allows cash-flow classes to take part in the Acyclic Visitor pattern ([Martin, 1997](#)); an example of its use will be provided in [a later section](#).

In earlier versions of the library, all these methods were declared in the `CashFlow` class. Later on, the date-related methods were moved into a separate class, `Event`, from which `CashFlow` inherits<sup>2</sup> and which is reused in other parts of the code. The interfaces of the two classes are shown in the listing below.

Interfaces of the `Event` and `CashFlow` classes.

---

```
class Event : public Observable {
public:
    virtual Date date() const = 0;
    bool hasOccurred(const Date &d) const;
    virtual void accept(AcyclicVisitor&);
};

class CashFlow : public Event {
public:
    virtual Real amount() const = 0;
    virtual void accept(AcyclicVisitor&);
};
```

---

<sup>1</sup>I trust I’m not revealing any secret business practice.

<sup>2</sup>Ok, so we didn’t start from the very top of the hierarchy. It’s still the top of the cash-flow related classes, though. I’m not here to con you, you know.

The library provides a simple implementation of the interface in the aptly named `SimpleCashFlow` class. It's boring enough that I won't show it here; it takes a date and an amount to be paid, and returns them from the `date` and `amount` methods, respectively. To find more interesting classes, we have to turn to interest-rate coupons—the subject of next section.

### Aside: late payments.

The implementation of the `Event::hasOccurred` method is simple enough: a date comparison. However, what should it return when the cash-flow date and the evaluation date are the same—or in other words, should today's payments be included in the present value of an instrument? The answer is likely to depend on the conventions of any given desk. QuantLib lets the user make his choice by means of a few global settings; the choice can be overridden at any given time by passing the appropriate boolean flag to `hasOccurred`.

## 4.2 Interest-rate coupons

The `Coupon` class (shown in the next listing) can be used as a parent class for any cash-flow that accrues an interest rate over a given period and with a given day-count convention. Of course, it is an abstract base class; it defines additional interface methods for any such cash flow, and implements a few concrete methods dealing with date calculations.

Interface of the `Coupon` class.

---

```
class Coupon : public CashFlow {
public:
    Coupon(Real nominal,
           const Date& paymentDate,
           const Date& accrualStartDate,
           const Date& accrualEndDate,
           const Date& refPeriodStart = Date(),
           const Date& refPeriodEnd = Date());

    Date date() const {
        return paymentDate_;
    }

    Real nominal() const {
        return nominal_;
    }
    const Date& accrualStartDate() const; // similar to the above
    const Date& accrualEndDate() const;
```

```

const Date& referencePeriodStart() const;
const Date& referencePeriodEnd() const;
Time accrualPeriod() const {
    return dayCounter().yearFraction(accrualStartDate_,
                                    accrualEndDate_,
                                    refPeriodStart_,
                                    refPeriodEnd_);
}
Integer accrualDays() const; // similar to the above

virtual Rate rate() const = 0;
virtual DayCounter dayCounter() const = 0;
virtual Real accruedAmount(const Date&) const = 0;

virtual void accept(AcyclicVisitor&);

protected:
    Real nominal_;
    Date paymentDate_, accrualStartDate_, accrualEndDate_,
        refPeriodStart_, refPeriodEnd_;
};

```

---

The abstract interface includes a `rate` method, which in derived classes will return the interest rate accrued by the coupon; and the `dayCounter` and `accruedAmount` methods, which return, respectively, the day-count convention used for accrual and the cash amount accrued until the given date.

The choice to declare the `rate` method as purely abstract seems obvious enough. However, the same doesn't hold for the other two methods. As for `dayCounter`, one could make a case for storing the day counter as a data member of the `Coupon` class; as discussed in [the previous chapter](#), this is what we did for the `TermStructure` class. Furthermore, the `nominal` method (which is conceptually similar) is not abstract and is based on a corresponding data member. As I'm all for abstract interfaces, I don't complain—well, apart from playing the devil's advocate here for illustration's purposes. But I admit that the asymmetry is somewhat disturbing.

The `accruedAmount` method is another matter; it is abstract for the wrong reason. It could have a default implementation in terms of the `rate` method, whose result would be multiplied by the notional and the accrual time up to the given date. To make it abstract was a choice *a posteriori*, due to the fact that a few derived classes define the `rate` method in terms of the `amount` method instead of the other way around. In such classes, `accruedAmount` is defined in terms of `amount`, on the dubious grounds that this might be more efficient. However, the correct choice would be to add the default implementation to the `Coupon` class and override it when (and if) required. We might do this in a future release.<sup>3</sup>

<sup>3</sup>The same could be said for the `amount` method; it could, too, have a default implementation.

The rest of the interface is made of concrete methods. The constructor takes a set of data and stores them in data members; the data include the nominal of the coupon, the payment date, and the dates required for calculating the accrual time. Usually, such dates are just the start and end date of the accrual period. Depending on the chosen day-count convention, two more dates (i.e., the start and end date of a reference period) might be needed; see [appendix A](#) for details.

For each of the stored data, the `Coupon` class defines a corresponding inspector; in particular, the one which returns the payment date implements the `date` method required by the `CashFlow` interface. Furthermore, the `accrualPeriod` and `accrualDays` methods are provided; as shown in the listing, they use the given day-count convention and dates to implement the corresponding calculations.

Two notes before proceeding. The first is that, as for the `dayCounter` method, we had an alternative here between storing the relevant dates and declaring the corresponding methods as abstract. As I said, I'm all for abstract interfaces; but in this case, a bit of pragmatism suggested that it probably wasn't a good idea to force almost every derived class to store the dates as data members and implement the same inspectors.<sup>4</sup> If you like, that's yet another hint that we should make `dayCounter` a concrete method.

The second note: exposing the dates through the corresponding inspectors is obviously the right thing to do, as the information might be needed for reporting or all kind of purposes. However, it might also give one the idea of using them for calculations, instead of relying on the provided higher-level methods such as `accrualPeriod`. Of course, it's not possible to restrict access, so all we can do is warn against it.

And finally—no, I haven't forgot to describe the `accept` method. I'll get back to it later on, as we tackle the Visitor pattern.

## 4.2.1 Fixed-rate coupons

Let's now turn to concrete classes implementing the `Coupon` interface. The simplest is of course the one modeling a fixed-rate coupon; it is called `FixedRateCoupon` and its implementation is sketched in the listing below. Actually, the current implementation is not the very simplest: although it started as a simply-compounding coupon, it was later generalized (by a user who needed it; as you know, premature generalization is evil) to support different compounding rules.

---

<sup>4</sup>The only derived classes that wouldn't need to store the coupon dates as data members would probably be those decorating an existing coupon.

Sketch of the `FixedRateCoupon` class.

---

```

class FixedRateCoupon : public Coupon {
    public:
        FixedRateCoupon(Real nominal,
                        const Date& paymentDate,
                        Rate rate,
                        const DayCounter& dayCounter,
                        const Date& accrualStartDate,
                        const Date& accrualEndDate,
                        const Date& refPeriodStart = Date(),
                        const Date& refPeriodEnd = Date())
            : Coupon(nominal, paymentDate,
                      accrualStartDate, accrualEndDate,
                      refPeriodStart, refPeriodEnd),
            rate_(InterestRate(rate,dayCounter,Simple)),
            dayCounter_(dayCounter) {}

        Real amount() const {
            return nominal() *
                (rate_.compoundFactor(accrualStartDate_,
                                      accrualEndDate_,
                                      refPeriodStart_,
                                      refPeriodEnd_) - 1.0);
        }

        Rate rate() const { return rate_; }
        DayCounter dayCounter() const { return dayCounter_; }
        Real accruedAmount(const Date&) const; // similar to amount

    private:
        InterestRate rate_;
        DayCounter dayCounter_;
};
```

---

The constructor takes the arguments required by the `Coupon` constructor, as well as the rate to be paid and the day-count convention to be used for accrual. The constructor in the listing takes a simple rate; another constructor, not shown here for brevity, takes an `InterestRate` instance instead. The nominal and dates are forwarded to the `Coupon` constructor, while the other arguments are stored in two corresponding data members; in particular, the rate (passed as a simple floating-point number) is used to instantiate the required `InterestRate`.

Part of the required `Coupon` interface (namely, the `rate` and `dayCounter` methods) is easily implemented by returning the stored values. The remaining methods—`amount` and `accruedAmount`—are implemented in terms of the available information. The amount is obtained by multiplying the nominal by the rate, compounded over the coupon life, and subtracting the nominal in order to

yield only the accrued interest. The accrued amount is obtained in a similar way, but with a different accrual period.

One final note to end this subsection. I mentioned earlier on that we might include in the base Coupon class a default implementation for the `amount` method; as you have already guessed, it would multiply the nominal by the rate and the accrual time. Well, that seemingly obvious implementation already breaks in this simple case—which seems to cast a reasonable doubt about its usefulness. Software design is never easy, is it?

## 4.2.2 Floating-rate coupons

The `FloatingRateCoupon` class is emblematic of the life of most software. It started simple, became more complex as time went by, and might now need some refactoring; its current implementation (sketched in the following listing) has a number of issues that I'll point out as I describe it.<sup>5</sup>

Sketch of the `FloatingRateCoupon` class.

---

```
class FloatingRateCoupon : public Coupon, public Observer {
public:
    FloatingRateCoupon(
        const Date& paymentDate,
        const Real nominal,
        const Date& startDate,
        const Date& endDate,
        const Natural fixingDays,
        const shared_ptr<InterestRateIndex>& index,
        const Real gearing = 1.0,
        const Spread spread = 0.0,
        const Date& refPeriodStart = Date(),
        const Date& refPeriodEnd = Date(),
        const DayCounter& dayCounter = DayCounter(),
        bool isInArrears = false);

    Real amount() const;
    Rate rate() const;
    Real accruedAmount(const Date&) const;
    DayCounter dayCounter() const;

    const shared_ptr<InterestRateIndex>& index() const;
    Natural fixingDays() const;
    Real gearing() const;
    Spread spread() const;
    virtual Date fixingDate() const;
```

<sup>5</sup>Unless they can be fixed (or deprecated and replaced) without breaking backward compatibility, we'll have to live with such shortcomings.

```

virtual Rate indexFixing() const;
virtual Rate convexityAdjustment() const;
virtual Rate adjustedFixing() const;
bool isInArrears() const;

void update();
virtual void accept(AcyclicVisitor&);

void setPricer(const shared_ptr<FloatingRateCouponPricer>&);
shared_ptr<FloatingRateCouponPricer> pricer() const;
protected:
    Rate convexityAdjustmentImpl(Rate fixing) const;
    // data members
};

FloatingRateCoupon::FloatingRateCoupon(
    const Date& paymentDate, const Real nominal,
    ...other arguments...)
: Coupon(nominal, paymentDate,
    startDate, endDate, refPeriodStart, refPeriodEnd),
/* `store the other data members` */
registerWith(index_);
registerWith(Settings::instance().evaluationDate());
}

Real FloatingRateCoupon::amount() const {
    return rate() * accrualPeriod() * nominal();
}

Rate FloatingRateCoupon::rate() const {
    pricer_->initialize(*this);
    return pricer_->swapletRate();
}

Date FloatingRateCoupon::fixingDate() const {
    Date d = isInArrears_ ? accrualEndDate_ : accrualStartDate_;
    return index_->fixingCalendar().advance(
        d, -fixingDays_, Days, Preceding);
}

Rate FloatingRateCoupon::indexFixing() const {
    return index_->fixing(fixingDate());
}

```

---

```

Rate FloatingRateCoupon::adjustedFixing() const {
    return (rate()-spread())/gearing();
}

Rate FloatingRateCoupon::convexityAdjustmentImpl(Rate f) const {
    return (gearing() == 0.0 ? 0.0 : adjustedFixing()-f);
}

Rate FloatingRateCoupon::convexityAdjustment() const {
    return convexityAdjustmentImpl(indexFixing());
}

```

---

The first issue might be the name itself: `FloatingRateCoupon` suggests a particular type of coupon, i.e., one that pays some kind of LIBOR rate. However, the class is more general than this and can model coupons paying different kind of rates—CMS or whatnot. Unfortunately, other names might be even worse; for instance, the one I briefly considered while writing this paragraph (`VariableRateCoupon`) suggests that the definition of the rate might change besides the value, which is not the case. All in all, I don't think there's any point in changing it now.

But enough with my ramblings; let's look at the implementation. The constructor takes (and forwards to the base-class constructor) the dates and notional needed by the `Coupon` class, a day counter, and a number of arguments related to the interest-rate fixing. These include an instance of the `InterestRateIndex` class taking care of the rate calculation<sup>6</sup> as well as other details of the fixing; namely, the number of fixing days, whether or not the rate is fixed in arrears, and an optional gearing and spread. The arguments that are not passed to the `Coupon` constructor are stored in data members; moreover, the instance registers as an observer of its interest-rate index and of the current evaluation date. As we will see shortly, the `fixingDays` and `inArrears` arguments are used to determine the fixing date. When given, the gearing and spread cause the coupon to pay a rate  $R = g \cdot F + s$  where  $g$  is the gearing,  $F$  is the fixing of the underlying index, and  $s$  is the spread.<sup>7</sup>

This choice of the constructor signature (and therefore, of the stored data members) implies a constraint on the kinds of coupon that can be adequately modeled by this class. We are limiting them to those whose rate is based on the fixing of a single index; others, such as those paying the spread between two rates, are excluded. True, they can be forcibly bolted on the `FloatingRateCoupon` class by creating some kind of spread class and inheriting it from `InterestRateIndex`; but this would somewhat break the mapping between the financial concepts being modeled and the class hierarchy, since the spread between two indexes is not itself an index (or at least, it is not usually considered one).

Other methods implement the required `CashFlow` and `Coupon` interfaces. Curiously enough, even

---

<sup>6</sup>See [appendix A](#) for details on `InterestRateIndex`. For the purpose of this section, it is enough to note that it can retrieve past index fixings and forecast future ones.

<sup>7</sup>In principle, one could model reverse-floater coupons by passing a negative gearing; but in practice, this is not advisable as it would neglect the implicit floor at zero that such coupons usually sport.

though floating-rate coupons are more complex than fixed-rate ones, the `amount` method has the simpler implementation: it multiplies the rate by the accrual time and the nominal. This seems to support moving it to the `Coupon` class. The `accrualAmount` method has a similar implementation (not shown in the listing) with a different accrual time. For the time being, I'll skip the `rate` method; we'll come back to its implementation in a short while.

Next come a number of inspectors. Besides `dayCounter` (which is required by the `Coupon` interface) we have those returning the parameters specific to floating-rate coupons, such as `index`, `fixingDays`, `gearing`, and `spread`. Finally, a number of methods are defined which implement some business logic.

The first two methods shown are `fixingDate` and `indexFixing`. They are both straightforward enough. The `fixingDate` method checks whether or not the coupon fixes in arrears, chooses a reference date accordingly (the end date of the coupon when in arrears, the start date otherwise) and moves it backward for the required fixing days; holidays are skipped according to the index calendar. The `indexFixing` method asks the stored index for its fixing at the relevant date.

Although the implementations of both methods are, as I said, straightforward, there is an issue with their signature. Just as the constructor assumes a single index, these two methods assume a single index fixing. This results in a loss of generality; for instance, it excludes coupons that pay the average fixing of an index over a set of several fixing dates, or that compounds rates over a number of sub-periods of the coupon life.

The last methods deal with the convexity adjustment (if any) to apply to the index fixing. The `adjustedFixing` method is written in terms of the `rate` method and inverts the  $R = g \cdot F + s$  formula to return the adjusted fixing  $\hat{F} = (R - s)/g$ . Its implementation is rather fragile, since it depends on the previous relationship between the rate and the fixing. If a derived class were to modify it (for instance, by adding a cap or floor to the paid rate—which has happened already, as we'll see later) this method should be modified accordingly. Unfortunately, this is left to the programmer; there's no language feature that can force one to override two methods at the same time.

The `convexityAdjustment` method returns the adjustment alone by taking the difference between the original fixing and the adjusted one. It does this by delegating to a protected `convexityAdjustmentImpl` method; this is a leftover of a previous implementation, in which the presence of a separate method allowed us to optimize the calculation. This is no longer needed in the current implementation; for the sake of simplicity, the protected method might be inlined into the public one and removed.

### Aside: keeping one's balance.

The assumptions built into the `FloatingRateCoupon` class (single index, single fixing date) are, of course, unfortunate. However, it is caused by the need to balance generality and usefulness in the class interface; the more general a class is, the fewer inspectors it can have and the less information it can return. The problem could be solved in part by adding more levels of inheritance (we might do that, if the need arises); but this adds complexity and brings its own problems. The current implementation of `FloatingRateCoupon` is probably not the best compromise; but it's one we can

live with for the time being.

Back to the `rate` method. In previous versions of the library, it was implemented as you might have expected—something like

```
Rate f = index_->fixing(fixingDate());
return gearing_ * (f + convexityAdjustment(f)) + spread_;
```

based on the fixing provided by the stored `InterestRateIndex` instance.<sup>8</sup> As much as I liked its simplicity, that implementation had to be changed when a new requirement came in: namely, that floating-rate coupons may be priced in several different ways. This was an issue that we had already faced with the `Instrument` class (see [chapter 2](#) for details).

Like we did for `Instrument`, we used the Strategy pattern to code our solution. However, the implementation we chose in this case was different. On the one hand, the context was more specific; we knew what kind of results we wanted from the calculations. On the other hand, `FloatingRateCoupon` had a richer interface than `Instrument`. This allowed us to avoid the opaque argument and result structures used in the implementation of the `PricingEngine` class.

The resulting `FloatingRateCouponPricer` class is sketched in the next listing. It declares methods for returning a number of rates: `swapletRate` returns the coupon rate, adjusted for convexity, gearing and spread (if any); `capletRate` returns the adjusted rate paid by a cap on the index fixing; and `floorletRate` returns does the same for a floor.<sup>9</sup>

Sketch of the `FloatingRateCouponPricer` class.

---

```
class FloatingRateCouponPricer: public virtual Observer,
                                public virtual Observable {
public:
    virtual ~FloatingRateCouponPricer();
    virtual void initialize(const FloatingRateCoupon&) = 0;
    virtual Rate swapletRate() const = 0;
    virtual Rate capletRate(Rate effectiveCap) const = 0;
    virtual Rate floorletRate(Rate effectiveFloor) const = 0;
    // other methods
};
```

---

The `initialize` method causes the pricer to store a reference to the passed coupon that will provide any information not passed to the other methods as part of the argument list.

<sup>8</sup>As you might have noted, the current `convexityAdjustment` implementation would cause an infinite recursion together with the code above. Its implementation was also different at that time and returned an explicit calculation of the adjustment.

<sup>9</sup>The class declares other methods, not shown here.

The `FloatingRateCoupon` class defines a `setPricer` method that takes a pricer instance and stores it into the coupon.<sup>10</sup> In the `rate` method, the stored pricer is initialized with the current coupon and the calculated rate is returned.

You might have noted that initialization is performed in the `rate` method, rather than in the `setPricer` method. This is done because the same pricer instance can be used for several coupons; therefore, one must make sure that the current coupon is stored each time the pricer is asked for a result. A previous calculation might have stored a different reference—which by now might even be dangling.

Of course, this implementation is not entirely satisfactory (and furthermore, it hinders parallelism, which is becoming more and more important since the last couple of years; as of now, we cannot pass the same pricer to several coupons and evaluate them simultaneously). It might have been a better design to define the pricer methods as, for instance,

```
Rate swapletRate(const FloatingRateCoupon& coupon);
```

and avoid the repeated call to `initialize`. Once again, I have to admit that we are guilty of premature optimization. We thought that, when `swapletRate` and other similar methods were called in sequence (say, to price a floored coupon), a separate initialization method could precompute some accessory quantities that would be needed in the different rate calculations; and we let that possibility drive the design. As it turned out, most pricers do perform some computations in their `initialize` method; but in my opinion, not enough to justify doing a Texas two-step to calculate a rate.

A refreshing remark to end this section. If you want to implement a specific floating-rate coupon but you don't care for the complexity added by the pricer machinery, you can inherit your class from `FloatingRateCoupon`, override its `rate` method to perform the calculation directly, and forget about pricers. They can be added later if the need arises.

### 4.2.3 Example: LIBOR coupons

The most common kind of floating-rate coupon is the one paying the rate fixed by a LIBOR index. A minimal implementation of such a coupon is shown in the following listing.

---

<sup>10</sup>The implementation is not as simple as that, but we can skip the details here.

Minimal implementation of the `IborCoupon` class.

---

```
class IborCoupon : public FloatingRateCoupon {
public:
    IborCoupon(const Date& paymentDate,
               const Real nominal,
               const Date& startDate,
               const Date& endDate,
               const Natural fixingDays,
               const shared_ptr<IborIndex>& index,
               const Real gearing = 1.0,
               const Spread spread = 0.0,
               const Date& refPeriodStart = Date(),
               const Date& refPeriodEnd = Date(),
               const DayCounter& dayCounter = DayCounter(),
               bool isInArrears = false)
        : FloatingRateCoupon(paymentDate, nominal,
                             startDate, endDate,
                             fixingDays, index, gearing, spread,
                             refPeriodStart, refPeriodEnd,
                             dayCounter, isInArrears) {}
};
```

---

The only required member is the constructor. It takes the same arguments as `FloatingRateCoupon`, but specializes the type of the passed index; it constrains this argument to be a pointer to an instance of the `IborIndex` class,<sup>11</sup> rather than the more generic `InterestRateIndex`. All arguments are simply forwarded to the constructor of the parent class; `index` needs no particular treatment, since shared pointers to a derived class can be implicitly upcast to pointers to its base class.

Of course, the real pricing work will be done by pricers. The next listing shows a partial implementation of one of them, namely, the `BlackIborCouponPricer` class; the *Black* part of the name refers to the Black model used for calculating the adjustment (if any) and possible caps or floors on the index fixing.

---

<sup>11</sup>The *Ibor* name was generalized from the common suffix of EURIBOR, LIBOR, and a number of like indexes.

Partial implementation of the `BlackIborCouponPricer` class.

---

```

class BlackIborCouponPricer : public FloatingRateCouponPricer {
public:
    BlackIborCouponPricer(
        const Handle<OptionletVolatilityStructure>& v =
            Handle<OptionletVolatilityStructure>())
        : capletVol_(v) {
            registerWith(capletVol_);
    }
    void initialize(const FloatingRateCoupon& coupon) {
        coupon_ = dynamic_cast<const IborCoupon*>(&coupon);
        gearing_ = coupon_->gearing();
        spread_ = coupon_->spread();
    }
    Rate swapletRate() const {
        return gearing_ * adjustedFixing() + spread_;
    }
    // other methods, not shown
protected:
    virtual Rate adjustedFixing(
        Rate fixing = Null<Rate>()) const {

        if (fixing == Null<Rate>())
            fixing = coupon_->indexFixing();

        Real adjustement = 0.0;
        if (!coupon_->isInArrears()) {
            adjustement = 0.0;
        } else {
            QL_REQUIRE(!capletVolatility().empty(),
                       "missing optionlet volatility");
            adjustment = // formula implementation, not shown
        }

        return fixing + adjustement;
    }
    const IborCoupon* coupon_;
    Real gearing_;
    Spread spread_;
    Handle<OptionletVolatilityStructure> capletVol_;
};
```

---

The constructor takes a handle to a term structure that describes the volatility of the LIBOR rate, stores it, and registers with it as an observer. If the coupon rate is not fixed in arrears, and if no cap or floor is defined, the volatility is not used; in that case, the handle can be empty (or can be omitted from the constructor call).

The `initialize` method checks that the passed coupon has the correct type by downcasting it to `IborCoupon`. If the cast succeeds, a few results can then be precomputed. Here—more for illustration than for anything else—we store the coupon gearing and spread in two data members; this will allow to write the remaining code more concisely. The `swapletRate` method simply applies the stored gearing and spread to the adjusted rate, whose calculation is delegated to the `adjustedFixing` method. If the coupon fixes at the beginning of its tenor, the adjusted rate is just the fixing of the underlying index. If the coupon fixes in arrears, a convexity adjustment is calculated and added to the fixing; I don't show the formula implementation here, but only the check for a valid caplet volatility. Other methods such as `capletRate` are not shown here; you can read their implementation in the library.

This would be all (and would make for a clean but very short example) were it not for those meddling kids—namely, for the common practice of using par coupons to price floating-rate notes. The library should allow one to choose whether or not to follow this practice; and as usual, there's at least a couple of ways to do it.

One possible way (shown in the listing that follows) is to bypass the pricer machinery and override the `rate` method for LIBOR coupons. If par coupons are enabled by setting a compilation flag, and if the coupon is not in arrears (in which case we still need a pricer), a built-in par-rate calculation is used; otherwise, the method call is forwarded to the base-class method, which in turn delegates the calculation to the pricer.

Extension of the `IborCoupon` class to support par coupons.

---

```
class IborCoupon : public FloatingRateCoupon {
public:
    // constructor as before

    Rate rate() const {

        #ifndef QL_USE_INDEXED_COUPON
        if (!isInArrears()) {
            Rate parRate = // calculation, not shown
            return gearing() * parRate + spread();
        }
        #endif

        return FloatingRateCoupon::rate();
    }
};
```

---

This works, but it has disadvantages. On the one hand, one can still set pricers to the coupon, but with no observable effect. On the other hand, different index fixings would be used for in-arrears and not in-arrears coupons fixing on the same date, which could throw off parity relationships or, worse, hedges.

A probably better way would be to implement a par-coupon pricer explicitly. As shown in the next listing, this can be done by inheriting from the Black pricer described earlier. The derived pricer just needs to override the `adjustedFixing` method; instead of asking the stored index for its fixing, it would extract its risk-free curve and use it to calculate the par rate. This is somewhat cleaner than overriding `rate` in the coupon class, doesn't suffer from the same disadvantages, and allows one to make the choice at run time.

Sketch of the `ParCouponPricer` class.

---

```
class ParCouponPricer : public BlackIborCouponPricer {
public:
    // constructor, not shown

protected:
    virtual Rate adjustedFixing(
        Rate fixing = Null<Rate>() const {

    if (fixing == Null<Rate>()) {
        Handle<YieldTermStructure> riskFreeCurve =
            index->termStructure();
        fixing = // calculation of par rate, not shown
    }

    return BlackIborCouponPricer::adjustedFixing(fixing);
}
};
```

---

This approach still has a problem, though; the `convexityAdjustment` method will return the difference between the par fixing and the index fixing, which is not due to convexity effects.<sup>12</sup> Unfortunately, it is not clear how to fix this—or at least, it is not clear to me—although a few possibilities come to mind. One is to override `convexityAdjustment` to detect par coupons and return a null adjustment, but it wouldn't work for in-arrears coupons. Another is to rename the method to `adjustment` and make it more generic, but this would lose information. The best one is probably to delegate the calculation of the convexity adjustment to the pricer; this might also help to overcome the fragility issues that I mentioned [earlier](#) when I described the `adjustedFixing` method.

The current way? It's a bit of a bummer. Read the aside below.

---

<sup>12</sup>The same problem must be faced when overriding the coupon's `rate` method.

### Aside: breach of contract.

The current implementation of `IborCoupon` enables par coupons by overriding the `indexFixing` method and making it return the par rate. It works, in a way; the coupon methods return the correct amount, and even the expected null convexity adjustment (the latter doesn't hold for other implementations and might be the reason of this choice).

However, the implementation is wrong, since it breaks the semantics of the method; it no longer returns the index fixing. A future release should change the code so that the whole class interface works as declared.

#### 4.2.4 Example: capped/floored coupons

Caps and floors are features that can be applied to any kind of floating-rate coupon. In our framework, that means that we'll want to add the features, possibly in a generic way, to a number of classes derived from `FloatingRateCoupon`.

This requirement suggests some flavor of the Decorator pattern. The implementation of the resulting `CappedFlooredCoupon` class is shown in the listing below.

Implementation of the `CappedFlooredCoupon` class.

---

```

class CappedFlooredCoupon : public FloatingRateCoupon {
    public:
        CappedFlooredCoupon(
            const shared_ptr<FloatingRateCoupon>& underlying,
            Rate cap = Null<Rate>(),
            Rate floor = Null<Rate>());
        Rate rate() const;
        Rate convexityAdjustment() const;

        bool isCapped() const { return isCapped_; }
        bool isFloored() const { return isFloored_; }
        Rate cap() const;
        Rate floor() const;
        Rate effectiveCap() const;
        Rate effectiveFloor() const;

        virtual void accept(AcyclicVisitor&);

        void setPricer(
            const shared_ptr<FloatingRateCouponPricer>& pricer);
    }
```

```

protected:
    shared_ptr<FloatingRateCoupon> underlying_;
    bool isCapped_, isFloored_;
    Rate cap_, floor_;
};

CappedFlooredCoupon::CappedFlooredCoupon(
    const shared_ptr<FloatingRateCoupon>& underlying,
    Rate cap, Rate floor)
: FloatingRateCoupon(underlying->date(),
                     underlying->nominal(),
                     ...other coupon parameters...),
underlying_(underlying),
isCapped_(false), isFloored_(false) {

    if (gearing_ > 0) {
        if (cap != Null<Rate>()) {
            isCapped_ = true;
            cap_ = cap;
        }
        if (floor != Null<Rate>()) {
            isFloored_ = true;
            floor_ = floor;
        }
    } else {
        if (cap != Null<Rate>()) {
            isFloored_ = true;
            floor_ = cap;
        }
        if (floor != Null<Rate>()) {
            isCapped_ = true;
            cap_ = floor;
        }
    }
    registerWith(underlying);
}

Rate CappedFlooredCoupon::rate() const {
    Rate swapletRate = underlying_->rate();
    Rate floorletRate = isFloored_ ?
        underlying_->pricer()->floorletRate(effectiveFloor()) :
        0.0;
    Rate capletRate = isCapped_ ?

```

```

        underlying_->pricer()->capletRate(effectiveCap());
        0.0;
    return swapletRate + floorletRate - capletRate;
}

Rate CappedFlooredCoupon::convexityAdjustment() const {
    return underlying_->convexityAdjustment();
}

Rate CappedFlooredCoupon::cap() const {
    if ( (gearing_ > 0) && isCapped_)
        return cap_;
    if ( (gearing_ < 0) && isFloored_)
        return floor_;
    return Null<Rate>();
}

Rate CappedFlooredCoupon::floor() const {
    if ( (gearing_ > 0) && isFloored_)
        return floor_;
    if ( (gearing_ < 0) && isCapped_)
        return cap_;
    return Null<Rate>();
}

Rate CappedFlooredCoupon::effectiveCap() const {
    if (isCapped_)
        return (cap_ - spread()) / gearing();
    else
        return Null<Rate>();
}

Rate CappedFlooredCoupon::effectiveFloor() const {
    if (isFloored_)
        return (floor_ - spread()) / gearing();
    else
        return Null<Rate>();
}

void CappedFlooredCoupon::setPricer(
    const shared_ptr<FloatingRateCouponPricer>& pricer) {
    FloatingRateCoupon::setPricer(pricer);
    underlying_->setPricer(pricer);
}

```

---

}

As we will see shortly, the class follows the canonical form of the Decorator pattern, storing a pointer to the base `FloatingRateCoupon` class, adding behavior where required and calling the methods of the stored object otherwise. However, note that C++, unlike other languages, provides another way to implement the Decorator pattern: namely, we could have combined templates and inheritance to write something like

```
template <class CouponType>
class CappedFloored : public CouponType;
```

This class template would be used to instantiate a number of classes (such as, e.g., `CappedFloored<IborCoupon>` or `CappedFloored<CmsCoupon>`) that would add behavior where required and call the methods of their respective base classes otherwise. At this time, I don't see any compelling reason for the template implementation to replace the existing one; but out of interest, during the description of `CappedFlooredCoupon`—which I'll start without further ado—I'll point out where the class template would have differed, for better or worse.

Not surprisingly, the constructor takes a `FloatingRateCoupon` instance to decorate and stores it in a data member. What might come as a surprise is that the constructor also extracts data from the passed coupon and copies them into its own data members by passing them to the base `FloatingRateCoupon` constructor. This is a departure from the usual implementation of the pattern, which would forward all method calls to the stored underlying coupon using code such as

```
Date CappedFlooredCoupon::startDate() const {
    return underlying_>startDate();
}
```

Instead, this implementation holds coupon state of its own and uses it to provide non-decorated behavior; for instance, the `startDate` method will return the copied data member. This causes some duplication of state, which is not optimal; but on the other hand, it avoids writing quite a few forwarding methods. The alternative would be to make all `Coupon` methods virtual and override them all.

The design might be a bit cleaner if we were to use template inheritance. In that case, the coupon to be decorated would be the decorator itself, and duplication of state would be avoided. However, the constructor would still need to copy state; we'd have to write it as

```
template <class CouponType>
CappedFlooredCoupon(const CouponType& underlying,
                     Rate cap, Rate floor)
: CouponType(underlying), ...
```

and not in the more desirable way, i.e., as a constructor taking the arguments required to instantiate the underlying; we'd have to build it externally to pass it, copy it, and throw it away afterwards. This is on the one hand, because of the so-called *forwarding problem* ((Dimov *et al*, 2002); and on the other hand, because coupons tend to take quite a few parameters, and we'd probably need to give CappedFlooredCoupon about a dozen template constructors to accept them all. Note that both problems are due to the fact that we're still using C++03 to support older compiler; they would be solved in C++11 by perfect forwarding (Hinnant *et al*, 2006) and variadic templates (Gregor, 2006), respectively, so we might revisit this design in the future.

The other constructor arguments are the cap and floor rates. If either of them is not needed, client code can pass `Null<Rate>()` to disable it. The passed rates are stored in two data members—with a couple of twists. The first is that, besides two data members for the cap and floor rates, the class also stores two booleans that tell whether they're enabled or disabled. This is somewhat different from what is done in several other places in the library; usually, we store the value and rely on comparison with `Null<Rate>()` to see whether it's enabled. Storing a separate boolean member might seem redundant, but it avoids using the null value as a magic number. It would be better yet to use a type, such as `boost::optional`, that is designed to store a possibly null value and has cleaner semantics. In fact, I'd like to delete the `Null` class and replace it with `optional` everywhere, but it would break too much client code at this time.

The second twist is that a cap might not be stored as a cap. If the gearing of the underlying is positive, all is as expected. If it's negative, the cap is stored in the `floor_` data member and vice versa. This was probably based on the fact that, when the gearing is negative, a cap on the coupon rate is in fact a floor on the underlying index rate. However, it might be confusing and complicates both the constructor and the inspectors. I think it would be better to store the cap and floor as they are passed.

The `rate` method is where the behavior of the underlying coupon is decorated. In our case, this means modifying the coupon rate by adding a floorlet rate (i.e., the rate that, accrued over the life of the coupon, gives the price of the floorlet) and subtracting a caplet rate. The calculation of the two rates is delegated to the pricer used for the underlying coupon, which requires as arguments the effective cap and floor rates; as we'll see shortly, these are the cap and floor rates that apply to the underlying index fixing rather than to the coupon rate.

The next method is `convexityAdjustment`. It forwards to the corresponding method in the underlying coupon, and returns the result without changes (incidentally, we'd get this behavior for free—i.e., without having to override the method—if we were to use template inheritance). This might look a bit strange, since applying the cap or floor to the coupon can change the rate enough to make the adjustment moot; but taking the cap and floor rates into account would change the meaning of the method, which was defined to return the adjustment of the underlying index fixing.

The `cap` and `floor` inspectors return the passed cap and floor, possibly performing an inverse flip

if a negative coupon gearing caused the constructor to flip the cap and floor data members. As I said, this might be improved by storing cap and floor unmodified. If necessary, the flip would be performed inside the `effectiveCap` and `effectiveFloor` method, that currently (the flip being already performed) only adjust the cap and floor for the coupon gearing and spread, thus returning the cap and floor on the underlying index fixing.

Finally, the `setPricer` method. In order to keep everything consistent, the passed pricer is set to both the underlying coupon and to this instance. It smells a bit: it's a bit of duplication of state, and a bit more work than would be necessary if we had written a pure Decorator—or if we had used template inheritance. But I've committed worse sins in my life as a developer; and such duplication as we have is confined in this single method.

At this point, the `CappedFlooredCoupon` class is complete. To wrap up the example, I'll mention that it's possible to inherit from it as a convenience for instantiating capped or floored coupons; one such derived class is shown in the next listing and specializes the base class to build capped or floored LIBOR coupons. The class only defines a constructor; instead of a built coupon instance, it takes the arguments needed to build the underling coupon, instantiates it, and passes it to the constructor of the base class. This way, it takes away a bit of the chore in creating a coupon. The implementation would be identical if `CappedFlooredCoupon` were a class template.

Implementation of the `CappedFlooredIborCoupon` class.

---

```
class CappedFlooredIborCoupon : public CappedFlooredCoupon {
public:
    CappedFlooredIborCoupon(
        const Date& paymentDate,
        const Real nominal,
        const Date& startDate,
        const Date& endDate,
        ...other IBOR-coupon arguments...
        const Rate cap = Null<Rate>(),
        const Rate floor = Null<Rate>(),
        ...remaining arguments...)
    : CappedFlooredCoupon(
        shared_ptr<FloatingRateCoupon>(new IborCoupon(...)),
        cap, floor) {}
};
```

---

Finally, a word on a possible alternative implementation. The class I described in this example applies a cap and a floor, both possibly null; which forces the class to contain a couple of booleans and some logic to keep track of what is enabled or disabled. This might be avoided if the cap and the floor were two separate decorators; each one could be applied separately, without the need of storing a null for the other one. However, two separate decorators with similar semantics would probably cause some duplicated code; and one decorator with a flag to select whether it's a cap or a floor would need

some logic to manage the flag. If you want to get some exercise by giving this implementation a try, you're welcome to contribute it to see how it compares to the current one.

### 4.2.5 Generating cash-flow sequences

Coupons seldom come alone. They come packed in legs—and with them, comes trouble for the library writer. Of course, we wanted to provide functions to build sequences of coupons based on a payment schedule; but doing so presented a few interface problems.

At first, we wrote such facilities in the obvious way; that is, as free-standing functions that took all the needed parameters (quite a few, in all but the simplest cases), built the corresponding leg, and returned it. However, the result was quite uncomfortable to call from client code: in order to build a floating-rate leg, the unsuspecting library user (that's you) had to write code like:

```
vector<shared_ptr<CashFlow>> leg =
    IborLeg(schedule, index,
        std::vector<Real>(1, 100.0),
        std::vector<Real>(1, 1.0),
        std::vector<Spread>(1, 0.004),
        std::vector<Rate>(1, 0.01),
        std::vector<Real>(),
        index->dayCounter(),
        true);
```

The above has two problems. The first is a common one for functions taking a lot of parameters: unless we look up the function declaration, it's not clear what parameters we are passing (namely, a notional of 100, a unit gearing, a spread of 40 bps, a floor of 1%, and fixings in arrears—did you guess them?) The problem might be reduced by employing named variables, such as:

```
std::vector<Real> notinals(1, 100.0);
std::vector<Real> unit_gearings(1, 1.0);
std::vector<Real> no_caps;
bool inArrears = true;
```

but this increases the verbosity of the code, as the above definitions must be added before the function call. Moreover, although several parameters have meaningful default values, they must be all specified if one wants to change a parameter that appears later in the list; for instance, specifying in-arrears fixings in the above call forced us to also pass a day counter and a null vector of cap rates.

The second problem is specific to our domain. Beside the coupon dates, other parameters can possibly change between one coupon and the next; e.g, the coupon notinals could decrease according to an amortizing plan, or a cap/floor corridor might follow the behavior of the expected forward rates. Since we need to cover the general case, the coupon builders must take all such parameters as vectors.

However, this forces us to verbosely instantiate vectors even when such parameters don't change—as it happens, most of the times.<sup>13</sup>

Both problems have the same solution. It is an idiom which has been known for quite a while in the C++ circles as the Named Parameter Idiom<sup>14</sup> and was made popular in the dynamic-language camp by Martin Fowler (of *Refactoring* fame) under the name of Fluent Interface ([Fowler, 2005](#)). In short: instead of a function, we'll use an object to build our coupons; and instead of passing the parameters together in a single call, we'll pass them one at a time to explicitly named methods. This will enable us to rewrite the function call in the previous example less verbosely and more clearly as

```
vector<shared_ptr<CashFlow>> leg =
    IborLeg(schedule, index)
    .withNotionals(100.0)
    .withSpreads(0.004)
    .withFloors(0.01)
    .inArrears();
```

A partial implementation of the `IborLeg` class is shown in the following listing.

Partial implementation of the `IborLeg` class.

---

```
class IborLeg {
public:
    IborLeg(const Schedule& schedule,
            const shared_ptr<IborIndex>& index);
    IborLeg& withNotionals(Real notional);
    IborLeg& withNotionals(const vector<Real>& notionals);
    IborLeg& withPaymentDayCounter(const DayCounter&);
    IborLeg& withPaymentAdjustment(BusinessDayConvention);
    IborLeg& withFixingDays(Natural fixingDays);
    IborLeg& withFixingDays(const vector<Natural>& fixingDays);
    IborLeg& withGearings(Real gearing);
    IborLeg& withGearings(const vector<Real>& gearings);
    IborLeg& withSpreads(Spread spread);
    IborLeg& withSpreads(const vector<Spread>& spreads);
    IborLeg& withCaps(Rate cap);
    IborLeg& withCaps(const vector<Rate>& caps);
    IborLeg& withFloors(Rate floor);
    IborLeg& withFloors(const vector<Rate>& floors);
    IborLeg& inArrears(bool flag = true);
    IborLeg& withZeroPayments(bool flag = true);
    operator Leg() const;
```

<sup>13</sup>By using overloading, we could accept both vectors and single numbers; but for  $N$  parameters, we'd need  $2^N$  overloads. This becomes unmanageable very quickly.

<sup>14</sup>It is described as an established idiom in *C++ FAQs* ([Cline et al, 1998](#)).

```

private:
    Schedule schedule_;
    shared_ptr<IborIndex> index_;
    vector<Real> notionalS_;
    DayCounter paymentDayCounter_;
    BusinessDayConvention paymentAdjustment_;
    vector<Natural> fixingDays_;
    vector<Real> gearings_;
    vector<Spread> spreads_;
    vector<Rate> caps_, floors_;
    bool inArrears_, zeroPayments_;

};

IborLeg::IborLeg(const Schedule& schedule,
                 const shared_ptr<IborIndex>& index)
: schedule_(schedule), index_(index),
  paymentDayCounter_(index->dayCounter()),
  paymentAdjustment_(index->businessDayConvention()),
  inArrears_(false), zeroPayments_(false) {}

IborLeg& IborLeg::withNotionals(Real notional) {
    notionalS_ = vector<Real>(1,notional);
    return *this;
}

IborLeg& IborLeg::withNotionals(const vector<Real>& notionalS) {
    notionalS_ = notionalS;
    return *this;
}

IborLeg& IborLeg::withPaymentDayCounter(
                 const DayCounter& dayCounter) {
    paymentDayCounter_ = dayCounter;
    return *this;
}

IborLeg::operator Leg() const {

    Leg cashflows = ...

    if (caps_.empty() && floors_.empty() && !inArrears_) {
        shared_ptr<FloatingRateCouponPricer> pricer(
            new BlackIborCouponPricer);
    }
}

```

```

        setCouponPricer(cashflows, pricer);
    }

    return cashflows;
}

```

---

The constructor takes just a couple of arguments, namely, those for which there's no sensible default and that can't be confused with other arguments of the same type. This restricts the set to the coupon schedule and the LIBOR index. The other data members are set to their default values, which can be either fixed ones or can depend on the passed parameters (e.g., the day counter for the coupon defaults to that of the index).

Any other leg parameters are passed through a setter. Each setter is unambiguously named, can be overloaded to take either a single value or a vector,<sup>15</sup> and return a reference to the `IborLeg` instance so that they can be chained. After all parameters are set, assigning to a `Leg` triggers the relevant conversion operator, which contains the actual coupon generation (in itself, not so interesting; it instantiates a sequence of coupons, each with the dates and parameters picked from the right place in the stored vectors).

Finally, let me add a final note; not on this implementation, but to mention that the Boost folks provided another way to solve the same problem. Managing once again to do what seemed impossible, their Boost Parameter Library gives one the possibility to use named parameters in C++ and write function calls such as

```

vector<shared_ptr<CashFlow>> leg =
    IborLeg(schedule, index,
        _notional = 100.0,
        _spread = 0.004,
        _floor = 0.01,
        _inArrears = true);

```

I've been tempted to use the above in QuantLib, but the existing implementation based on the Named Parameter Idiom worked and needed less work from the developers. Just the same, I wanted to mention the Boost solution here for its sheer awesomeness.

## 4.2.6 Other coupons and further developments

Of course, other types of coupons can be implemented besides those shown here. On the one hand, the ones I described just begin to explore interest-rate coupons. The library provides a few others in the same style, such as those paying a constant-maturity swap rate; their pricer merely implements a different formula. Others, not yet implemented, would need additional design; for example, an

---

<sup>15</sup>For  $N$  parameters we'll need  $2N$  overloads, which is much more manageable than the alternative.

interesting exercise you can try might be to generate a sequence of several cash flows obtaining their amounts from a single Monte Carlo simulation.<sup>16</sup>

On the other hand, the same design can be used for other kinds of coupons besides interest-rate ones; this was done, for instance, for inflation-rate coupons. Of course, this begs the question of whether or not the design can be generalized to allow for different kind of indexes. On the whole, I'm inclined to wait for a little more evidence (say, a third and so-far unknown kind of coupon) before trying to refactor the classes.

## 4.3 Cash-flow analysis

After having generated one or more legs, we naturally want to analyze them. The library provides a number of commonly used functions such as NPV, basis-point sensitivity, yield, duration, and others. The functions are defined as static methods of a `CashFlows` class so that they do not take up generic names such as `npv` in the main `QuantLib` namespace; the same could have been done by defining them as free functions into an inner `CashFlows` namespace.

The functions themselves, being the implementation of well-known formulas, are not particularly interesting from a coding point of view; most of them are simple loops over the cash-flow sequence, like the `npv` function sketched in the listing below. What's interesting is that, unlike `npv`, they might need to access information that is not provided by the generic `CashFlow` interface; or they might need to discriminate between different kinds of cash flows.

Sketch of the `CashFlows::npv` method.

---

```
Real CashFlows::npv(const Leg& leg,
                     const YieldTermStructure& discountCurve,
                     Date settlementDate) {
    ...checks...

    Real totalNPV = 0.0;
    for (Size i=0; i<leg.size(); ++i) {
        if (!leg[i]->hasOccurred(settlementDate))
            totalNPV += leg[i]->amount() *
                         discountCurve.discount(leg[i]->date());
    }
    return totalNPV;
}
```

---

One such function calculates the basis-point sensitivity of a leg. It needs to discriminate between coupons accruing a rate and other cash flows, since only the former contribute to the BPS. Then, it has to call the `Coupon` interface to obtain the data needed for the calculation.

<sup>16</sup>Hint: the pricers for the coupons would share a pointer to a single Monte Carlo model, as well as some kind of index (an integer would suffice) identifying the coupon and allowing to retrieve the correct payoff among the results.

Of course, this could be done by trying a `dynamic_pointer_cast` on each cash flow and adding terms to the result when the cast succeeds. However, casting is somewhat frowned upon in the high society of programming—especially so if we were to chain `if` clauses to discriminate between several cash-flow types.

Instead of explicit casts, we implemented the `bps` method by means of the Acyclic Visitor pattern (see appendix A for details). Mind you, this was not an obvious choice. On the one hand, explicit casts can litter the code and make it less clear. On the other hand, the Visitor pattern is one of the most debated, and it's definitely not a simple one. Depending on the function, either casts or a visitor might be the most convenient choice. My advice (which I'm going to disregard in the very next paragraph) is to have a strong bias towards simplicity.

In the case of `bps`, casts might in fact have sufficed; the resulting code would have been something like

```
for (Size i=0; i<leg.size(); ++i) {
    if (shared_ptr<Coupon> c =
        dynamic_pointer_cast<Coupon>(leg[i])) {
        // do something with c
    }
}
```

which is not that bad a violation of object-oriented principles—and, as you'll see in a minute, is simpler than the visitor-based implementation. One reason why we chose to use a visitor was to use `bps` as an example for implementing more complex functions, for which casts would add too much scaffolding to the code.<sup>17</sup>

The implementation of the `bps` method is sketched in the next listing. For the time being, I'll gloss over the mechanics of the Acyclic Visitor pattern (as I said, the details are in appendix A) and just describe the steps we've taken to use it.

Sketch of the `CashFlows::bps` method.

---

```
namespace {

    const Spread basisPoint_ = 1.0e-4;

    class BPSCalculator : public AcyclicVisitor,
                          public Visitor<CashFlow>,
                          public Visitor<Coupon> {
        public:
            BPSCalculator(const YieldTermStructure& discountCurve)
                : discountCurve_(discountCurve), result_(0.0) {}
```

---

<sup>17</sup>Other reasons might have included glamour or overthinking—I know I've been guilty of them before. Nowadays, I'd probably use a simple cast. But I think that having an example is still reason enough to keep the current implementation.

```

void visit(Coupon& c) {
    result_ += c.nominal() *
        c.accrualPeriod() *
        basisPoint_ *
        discountCurve_.discount(c.date());
}
void visit(CashFlow&) {}
Real result() const {
    return result_;
}
private:
    const YieldTermStructure& discountCurve_;
    Real result_;
};

}

Real CashFlows::bps(const Leg& leg,
                    const YieldTermStructure& discountCurve,
                    Date settlementDate) {
    ...checks...

    BPSCalculator calc(discountCurve);
    for (Size i=0; i<leg.size(); ++i) {
        if (!leg[i]->hasOccurred(settlementDate))
            leg[i]->accept(calc);
    }
    return calc.result();
}

```

---

Our visitor—the `BPSCalculator` class—inherits from a few classes. The first is the `AcyclicVisitor` class, needed so that our class matches the interface of the `CashFlow::accept` method. The others are instantiation of the `Visitor` class template specifying what different kind of cash flows should be processed. Then, the class implements different overloads of a `visit` method. The overload taking a `Coupon` will be used for all cash-flows inheriting from such class, and will access its interface to calculate the term to add to the result. The overload taking a `CashFlow` does nothing and will be used as a fallback for all other cash flows.

At this point, the class can work with the pattern implementation coded in the cash-flow classes. The `bps` method is written as a loop over the cash flows, similar to the one in the `npv` method. An instance of our visitor is created before starting the loop. Then, for each coupon, the `accept` method is called and passed the visitor as an argument; each time, this causes the appropriate overload of `visit` to be called, adding terms to the result for each coupon and doing nothing for any other cash

flow. At the end, the result is fetched from the visitor and returned.

As I said, it's not simple; the same code using a cast would probably be half as long. However, the scaffolding needed for implementing a visitor remains pretty much constant; functions that need to discriminate between several kinds of cash flow can follow this example and end up with a higher signal-to-noise ratio.

### 4.3.1 Example: fixed-rate bonds

In this example, I'll try and bolt the cash-flow machinery on the pricing-engine framework. The instrument I target is the fixed-rate bond; but hindsight being 20/20, I'll put most code in a base `Bond` class (as it turns out, most calculations are generic enough that they work for any bond). Derived classes such as `FixedRateBond` will usually just contain code for building their specific cash flows.

Naturally, users will expect to retrieve a great many values from a bond instance: clean and dirty prices, accrued amount, yield, whatever strikes a trader's fancy. The `Bond` class will need methods to return such values. Part of the resulting interface is shown in the listing below; you can look at the class declaration in the library for a full list of the supported methods.

Partial interface of the `Bond` class.

---

```
class Bond : public Instrument {
public:
    class arguments;
    class results;
    class engine;

    Bond(...some data...);

    bool isExpired() const;
    bool isTradable(Date d = Date()) const;
    Date settlementDate(Date d = Date()) const;
    Real notional(Date d = Date()) const;

    const Leg& cashflows() const;
    ...other inspector...

    Real cleanPrice() const;
    Real dirtyPrice() const;
    Real accruedAmount(Date d = Date()) const;
    Real settlementValue() const;

    Rate yield(const DayCounter& dc,
              Compounding comp,
              Frequency freq,
```

```

    Real accuracy = 1.0e-8,
    Size maxEvaluations = 100) const;

protected:
    void setupExpired() const;
    void setupArguments(PricingEngine::arguments*) const;
    void fetchResults(const PricingEngine::results*) const;

    Leg cashflows_;
    ...other data members...

    mutable Real settlementValue_;
};

class Bond : public Instrument {
public:
    ...other methods...

    Real cleanPrice(Rate yield,
                    const DayCounter& dc,
                    Compounding comp,
                    Frequency freq,
                    Date settlementDate = Date()) const;

    Rate yield(Real cleanPrice,
               const DayCounter& dc,
               Compounding comp,
               Frequency freq,
               Date settlementDate = Date(),
               Real accuracy = 1.0e-8,
               Size maxEvaluations = 100) const;
};

```

---

Now, if all the corresponding calculations were to be delegated to a pricing engine, this would make for a long but fairly uninteresting example. However, this is not the case. A very few calculations, such as the one returning the settlement value of the bond, are depending on the chosen model and will be performed by the pricing engine. Other calculations are independent of the model (in a sense—I'll explain presently) and can be performed by the `Bond` class itself, avoiding duplication in the possible several engines.

As a consequence, the set of data in the `Bond::results` class (and the corresponding `mutable` data members in the `Bond` class itself) is pretty slim; in fact, it consists only of the settlement value, i.e.,

the sum of future cash flows discounted to the settlement date of the bond.<sup>18</sup> The `settlementValue` method, shown in the next listing with the others I'll be discussing here, is like most methods that rely on a pricing engine: it triggers calculation if needed, checks that the engine assigned a value to the data member, and returns it.

Partial implementation of the `Bond` class.

---

```

Real Bond::settlementValue() const {
    calculate();
    QL_REQUIRE(settlementValue_ != Null<Real>(),
               "settlement value not provided");
    return settlementValue_;
}

Date Bond::settlementDate(Date d) const {
    if (d == Date())
        d = Settings::instance().evaluationDate();
    Date settlement =
        calendar_.advance(d, settlementDays_, Days);
    return std::max(settlement, issueDate_);
}

Real Bond::dirtyPrice() const {
    Real currentNotional = notional(settlementDate());
    if (currentNotional == 0.0)
        return 0.0;
    else
        return settlementValue()*100.0/currentNotional;
}

Real Bond::cleanPrice() const {
    return dirtyPrice() - accruedAmount(settlementDate());
}

Rate Bond::yield(const DayCounter& dc,
                 Compounding comp,
                 Frequency freq,
                 Real accuracy,
                 Size maxEvaluations) const {
    Real currentNotional = notional(settlementDate());
    if (currentNotional == 0.0)
        return 0.0;
}

```

---

<sup>18</sup>The NPV is distinct from the settlement value in that the former discounts the cash flows to the reference date of the discount curve.

```

return CashFlows::yield(cashflows(), dirtyPrice(),
    dc, comp, freq, false,
    settlementDate(), settlementDate(),
    accuracy, maxIterations);
}

```

---

Other calculations are of two or three kinds, all represented in the listing. Methods of the first kind return static information; they're not necessarily simple inspectors, but in any case the returned information is determined and doesn't depend on a given model. One such method, shown in the listing, is the `settlementDate` method which returns the settlement date corresponding to a given evaluation date. First, if no date is given, the current evaluation date is assumed; then, the evaluation date is advanced by the stored number of settlement days; and finally, we check that the result is not earlier than the issue date (before which the bond cannot be traded). Another such method, not shown, is the `notional` method which returns the notional of a bond (possibly amortizing) at a given date.

The second kind of methods return information such as the clean and dirty price of the bond. Of course such values depend on the model used, so that in principle they should be calculated by the engine; but in practice, they can be calculated from the settlement value independently of how it was obtained. One such method, shown in the listing, is the `dirtyPrice` method; it calculates the dirty price of the bond by taking the settlement value and normalizing it to 100 through a division by the current notional. If the current notional is null (and thus cannot divide the settlement value) then the bond is expired and the dirty price is also null. The `cleanPrice` method, also shown, takes the dirty price and subtracts the accrued amount to obtain the clean price. Finally, the `yield` method takes the dirty price, the cash flows, the settlement date, and the passed parameters and calls the appropriate method of the `CashFlows` class to return the bond yield.

A third kind of methods is similar to the second; they return results based on other values. The difference is simply that the input values are passed by the caller, rather than taken from engines: examples of such methods are the overloads of the `cleanPrice` and `yield` shown at the end of the `Bond` interface. The first one takes a `yield` and returns the corresponding clean price; the second one does the opposite. They, too, forward the calculation to a method of the `CashFlows` class.

The final step to obtain a working `Bond` class is to provide a pricing engine. A simple one is sketched in the listing that follows. Its constructor takes a discount curve and stores it; its `calculate` method passes the stored curve and the cash flows of the bond to the `CashFlows::npv` method. By passing as discount date the settlement date of the bond, it obtains its settlement value; by passing the reference date of the discount curve, its NPV.

Sketch of the `DiscountingBondEngine` class.

---

```
class DiscountingBondEngine : public Bond::engine {
public:
    DiscountingBondEngine(
        const Handle<YieldTermStructure>& discountCurve);
    void calculate() const {
        QL_REQUIRE(!discountCurve_.empty(),
                   "discounting term structure handle is empty");

        results_.settlementValue =
            CashFlows::npv(arguments_.cashflows,
                           **discountCurve_,
                           false,
                           arguments_.settlementDate);

        // same for `results_.value`, but discounting the cash flows
        // to the reference date of the discount curve.
    }
private:
    Handle<YieldTermStructure> discountCurve_;
};
```

---

All that remains is to provide the means to instantiate bonds—for the purpose of our example, fixed-rate bonds. The needed class is shown in the next listing. It doesn’t have much to do; its constructor takes the needed parameters and uses them to instantiate the cashflows. Apart for maybe a few inspectors, no other methods are needed as the `Bond` machinery can take over the calculations.

Implementation of the `FixedRateBond` class.

---

```
class FixedRateBond : public Bond {
public:
    FixedRateBond(Natural settlementDays,
                  const Calendar& calendar,
                  Real faceAmount,
                  const Date& startDate,
                  const Date& maturityDate,
                  const Period& tenor,
                  const std::vector<Rate>& coupons,
                  const DayCounter& accrualDayCounter,
                  BusinessDayConvention accrualConvention =
                      Following,
                  BusinessDayConvention paymentConvention =
                      Following,
```

```

    Real redemption = 100.0,
    const Date& issueDate = Date(),
    const Date& firstDate = Date(),
    const Date& nextToLastDate = Date(),
    DateGeneration::Rule rule =
        DateGeneration::Backward)
: Bond(...needed arguments...) {

    Schedule schedule = MakeSchedule()
        .from(startDate).to(maturityDate)
        .withTenor(tenor)
        .withCalendar(calendar)
        .withConvention(accrualConvention)
        .withRule(rule)
        .withFirstDate(firstDate)
        .withNextToLastDate(nextToLastDate);

    cashflows_ = FixedRateLeg(schedule)
        .withNotionals(faceAmount)
        .withCouponRates(coupons, accrualDayCounter)
        .withPaymentAdjustment(paymentConvention);

    shared_ptr<CashFlow> redemption(
        new SimpleCashFlow(faceAmount*redemption,
                            maturityDate));
    cashflows.push_back(redemption);
}
};

```

---

Now, to quote Craig Ferguson: what did we learn on the show tonight? The example showed how to code common calculations in an instrument class (here, the `Bond` class) independently of the pricing engine. The idea seems sound—the calculations should be replicated in each engine otherwise—but I should note that it comes with a disadvantage. In the design shown here (which is also the one currently implemented in the library) the calculations outside the pricing engine cannot use the caching mechanism described in [chapter 2](#). Thus, for instance, the current notional is calculated and the settlement value is normalized again and again every time the `dirtyPrice` method is called.

There would be a couple of ways out of this, neither exactly satisfactory. The first one would be to group the calculations together in the `performCalculation` method, after the engine worked its magic. Something like:

```
void Bond::performCalculations() const {
    Instrument::performCalculation();
    Real currentNotional = ...;
    dirtyPrice_ = settlementValue_*100.0/currentNotional;
    cleanPrice_ = ...;
    yield_ = ...;
    // whatever else needs to be calculated.
}
```

However, this has the problem that all calculations are performed, including slower ones such as the yield calculation (which needs iterative root solving). This would not be convenient if one, for instance, just wanted bond prices.<sup>19</sup>

A second alternative would be to add some kind of caching mechanism to each method. This would avoid unneeded calculations, but could get messy very quickly. It would require to store a separate observer for each method, register all of them with the appropriate observables, and check in each method whether the corresponding observer was notified. If this kind of performance was required, I'd probably do this on a per-method basis, and only after profiling the application.

All in all, though, I'd say that the current compromise seems acceptable. The repeated calculations are balanced by simpler (and thus more maintainable) code. If one were to need more caching, the exiting code can be patched to obtain it.

---

<sup>19</sup>Calculating all results might also be a problem for pricing engines. If performance is paramount—and, of course, depending on one's application—one might consider adding “light” versions of the existing engines that only calculate some results.

# 5. Parameterized models and calibration

Critics of the practice of calibration argue that its very existence is a sign of a problem. After all, if physicists had to recalibrate the universal constant of gravitation yearly, it would probably mean that the formula is invalid.<sup>1</sup>

For better or for worse, QuantLib supports calibration to market data because, well, that's what people need to do. Much like C++ or Smith & Wesson, we might add some safety but we assume that users know what they're doing, even if this gives them the possibility to shoot their foot off.

The calibration framework is one of the oldest parts of the library and has received little attention in the last few years; so it's likely that, as I write this chapter, I'll find and describe a number of things that could be improved—by breaking backward compatibility, I'm afraid, so they'll have to wait. In the meantime, you can learn from our blunders.

Onwards. The framework enables us to write code such as, for instance,

```
HullWhite model(termStructure);
Simplex optimizer(0.01);
model.calibrate(marketSwaptions,
                 optimizer,
                 EndCriteria(maxIterations, ...));
check(model.endCriteria());
// go on using the model
```

The above works because of the interplay of two classes called `CalibratedModel` and `CalibrationHelper`; the `HullWhite` class inherits from the former, while the elements of `marketSwaptions` are instances of a class that inherits from the latter.<sup>2</sup> Since they work together, describing either class before the other will cause some vagueness and hand-waving. Bear with me as I try to minimize the inconvenience (pun not intended).

## 5.1 The `CalibrationHelper` class

I'll describe the `CalibrationHelper` class first, since it depends only indirectly on the model (in fact, it doesn't use the model interface directly at all). It is shown in its entirety in the next listing: as seldom the case in our library, it is a pure interface.

---

<sup>1</sup>Or that there's something wrong with the idea of natural laws altogether, which is *way* scarier. This doesn't seem to be the case for physics. The jury is still out for quantitative finance.

<sup>2</sup>We're also using accessory classes, such as `Simplex`, that implement optimization methods; but I'll postpone their description to appendix A.

Interface of the `CalibrationHelper` class.

---

```
class CalibrationHelper {
public:
    virtual ~CalibrationHelper() {}
    virtual Real calibrationError() = 0;
};
```

---

The purpose of the class is similar—the name is a giveaway, isn’t it?—to that of the `BootstrapHelper` class, described in [chapter 3](#). It models a single quoted instrument (a “node” of the model, whatever that might be) and provides the means to calculate the instrument value according to the model and to check how far off it is from the market value. Actually, the value isn’t the only possibility; we’ll get to this in a bit.

Most derived classes, however, don’t inherit directly from `CalibrationHelper` but from its child `BlackCalibrationHelper`, which adds quite a bit of reusable behavior.<sup>3</sup> Its implementation is shown in the listing below.

Implementation of the `BlackCalibrationHelper` class.

---

```
class BlackCalibrationHelper : public CalibrationHelper,
                                public LazyObject {

public:
    enum CalibrationErrorType {
        RelativePriceError, PriceError, ImpliedVolError};

    BlackCalibrationHelper(
        const Handle<Quote>& volatility,
        const Handle<YieldTermStructure>& termStructure,
        CalibrationErrorType calibrationErrorType
            = RelativePriceError);

    void performCalculations() const {
        marketValue_ = blackPrice(volatility_->value());
    }
    virtual Real blackPrice(Volatility volatility) const = 0;
    Real marketValue() const {
        calculate(); return marketValue_;
    }

    virtual Real modelValue() const = 0;
    virtual Real calibrationError();
    void setPricingEngine(
```

---

<sup>3</sup>In past versions of the library, `BlackCalibrationHelper` was called `CalibrationHelper` and the latter didn’t exist.

```

        const shared_ptr<PricingEngine>& engine) {
    engine_ = engine;
}

Volatility impliedVolatility(Real targetValue,
                             Real accuracy,
                             Size maxEvaluations,
                             Volatility minVol,
                             Volatility maxVol) const;
virtual void addTimesTo(list<Time>& times) const = 0;

protected:
mutable Real marketValue_;
Handle<Quote> volatility_;
Handle<YieldTermStructure> termStructure_;
shared_ptr<PricingEngine> engine_;

private:
class ImpliedVolatilityHelper;
const CalibrationErrorType calibrationErrorType_;
};

class BlackCalibrationHelper::ImpliedVolatilityHelper {
public:
    ImpliedVolatilityHelper(const BlackCalibrationHelper&,
                           Real value);
    Real operator()(Volatility x) const {
        return value_ - helper_.blackPrice(x);
    }
    ...
};

Volatility BlackCalibrationHelper::impliedVolatility(
    Real targetValue, Real accuracy, Size maxEvaluations,
    Volatility minVol, Volatility maxVol) const {
    ImpliedVolatilityHelper f(*this,targetValue);
    Brent solver;
    solver.setMaxEvaluations(maxEvaluations);
    return solver.solve(f,accuracy,volatility_->value(),
                        minVol,maxVol);
}

Real BlackCalibrationHelper::calibrationError() {
    Real error;
}

```

```

switch (calibrationErrorType_) {
    case RelativePriceError:
        error = fabs(marketValue() - modelValue()) / marketValue();
        break;
    case PriceError:
        error = marketValue() - modelValue();
        break;
    case ImpliedVolError: {
        const Real modelPrice = modelValue();
        // check for bounds, not shown
        Volatility implied = this->impliedVolatility(
            modelPrice, 1e-12, 5000, 0.001, 10);
        error = implied - volatility_->value();
    }
    break;
default:
    QL_FAIL("unknown Calibration Error Type");
}
return error;
}

```

---

`BlackCalibrationHelper` also inherits from `LazyObject`, that you already know. The reason is that it might need some preliminary calculation: the target value of the optimization (say, the value) might not be available directly, for instance because the market quotes the corresponding implied volatility instead. The calculation to go from the one to the other must be done just once, before the calibration, and is done lazily as the market quote changes.

The constructor takes three arguments—each one maybe a bit less generic than I'd like, even though I only have minor complaints. The first argument is a handle to the quoted volatility; the assumption here is that, whatever the model is, that's how the market quotes the relevant instruments.

The second argument is a handle to a term structure, that we assumed to need for the calculations. That's true, but the curve is only ever used by derived classes, not here; so I prefer it to be declared there, along with any other data they might need. I'll deprecate and move this parameter in future releases.

Finally, the third argument specifies how the calibration error is defined. It's an enumeration that can take one of three values, meaning to take the relative error between the market price and the model price, or the absolute error between the prices, or the absolute error between the quoted volatility and the (Black) volatility implied by the model price. In principle, we might have used a Strategy pattern instead; but I'm not sure that the generalization is worth the added complexity, especially as I don't have a possible fourth case in mind.

The body of the constructor is not shown here for brevity, but it does the usual things: it stores its arguments in the corresponding data members and registers with those that might change.

As I said, the `LazyObject` machinery is used when market data change; accordingly, the required `performCalculations` method transforms the quoted volatility into a market price and stores it. The actual calculation depends on the particular instrument, so it's delegated to a purely virtual `blackPrice` method; the evident assumption is that a Black model was used to quote the market volatility. Finally, a `marketValue` method exposes the calculated price.

Note that, unfortunately, we need all three of the above methods. Yes, I know, it bugs me too. Obviously, `performCalculations` is required by the `LazyObject` interface; but what about `blackPrice` and `marketValue`? Can't we collapse them into one? Well, no. We want the `marketValue` inspector to be lazy, and therefore it must call `performCalculations`; thus, it can't be the same as `blackPrice`, which is called *by* the `performCalculations` method.<sup>4</sup>

The next set of methods deals with the model-based calculations which are executed during calibration. The purely virtual `modelValue`, when implemented in derived classes, must return the value of the instrument according to the model; the `calibrationError` method, that I'll describe in more detail later, returns some kind of difference between the market and model values; and the `setPricingEngine` brings the model into play.

The idea here is that the engine that we're storing has a pointer to the model, and can be used to price the instrument represented by the helper and thus give us a market value. All the helpers currently in the library implement the `modelValue` method as a straightforward translation of the idea: they set the given engine to the instrument they store, and ask it for its NPV (you'll see it spelled out later.)

In fact, the implementations are so similar that I wonder if we could have provided a common one in the base class. Had we added a pointer to an `Instrument` instance as a data member, the method would just be:

```
void BlackCalibrationHelper::modelValue() const {
    instrument_->setPricingEngine(engine_);
    return instrument_->NPV();
}
```

and with a bit more care, we could have set the pricing engine just once, at the beginning of the calibration, instead of each time the model value is recalculated. The downside of this would have been that the results of the methods would have been dependent on the order in which they were called; for instance, a call to the `modelValue` right after a call to `marketValue` might have returned the Black price if the latter had set a different engine to the instrument. According to Murphy's law, this would have bitten us back.

A last remark about the `setPricingEngine` method: when I re-read it, I first thought that there was a bug in it and that it should register with the engine. Actually, it shouldn't: the engine is used for the model value only, and must not trigger the `LazyObject` machinery that recalculates the market value.

---

<sup>4</sup>They also have a different interface, since `blackPrice` takes the volatility as an argument; but that could have been managed by giving it a default argument falling back to the stored volatility value.

The last two methods are utilities that can be used to help the calibration. The `impliedVolatility` method uses a one-dimensional solver to invert the `blackPrice` method; that is, to find the volatility that yields the corresponding Black price. Its implementation is shown in the second page of the listing, along with the sketch of an accessory inner class `ImpliedVolatilityHelper` that provides the objective function for the solver.

The `addTimesTo` method is to be used with tree-based methods (we'll get to those in [chapter 7](#)). It adds to the passed list a set of times that are of importance to the underlying instrument (e.g., payment times, exercise times, or fixing times) and therefore must be included in any time grid to be used. If I were to write it now, I'd just return the times instead of extending the given list, but that's a minor point. Another point is that, as I said, this method is tied to a particular category of models, that is, tree-based ones, and might not make sense for all helpers. Thus, I would provide an empty implementation so that derived classes don't necessarily have to provide one. However, I wouldn't try to move this method in some other class—say, a derived class meant to model helpers for tree-based models. On the one hand, it would add complexity and give almost no upside; and on the other hand, we can't even categorize helpers in this way: any given helper could be used with both types of models.

Finally, the listing shows the implementation of the `calibrationError` method. It is called by the calibration routine every time new parameters are set to the method, and returns an error that tells up how far we are from market data. The definition of "how far" is given by the stored enumeration value; if can be the relative difference of the model and market prices, their absolute difference, or the difference between the quoted volatility and the one implied by the model price.

### 5.1.1 Example: the Heston model

In this chapter, I'll use the Heston model as an example. Here, I'll describe the helper class; the model class will follow the discussion of the `CalibratedModel` class in [a later section](#).

The `HestonModelHelper` class is shown in the next listing. It models a European option, and right here we have a code smell; the name says nothing of the nature of the instrument, and instead it refers to a Heston model that is nowhere to be seen in the implementation of the helper. I, for one, didn't have the issue clear when this class was added.

Implementation of the `HestonModelHelper` class.

---

```
class HestonModelHelper : public BlackCalibrationHelper {
public:
    HestonModelHelper(
        const Period& maturity,
        const Calendar& calendar,
        const Real s0,
        const Real strikePrice,
        const Handle<Quote>& volatility,
        const Handle<YieldTermStructure>& riskFreeRate,
```

```

        const Handle<YieldTermStructure>& dividendYield,
        CalibrationErrorType errorType = RelativePriceError)
: BlackCalibrationHelper(volatility_, riskFreeRate_, errorType),
  dividendYield_(dividendYield),
  exerciseDate_(calendar.advance(
      riskFreeRate->referenceDate(), maturity)),
  tau_(riskFreeRate->dayCounter().yearFraction(
      riskFreeRate->referenceDate(), exerciseDate_)),
  s0_(s0), strikePrice_(strikePrice) {
    shared_ptr<StrikedTypePayoff> payoff(
        new PlainVanillaPayoff(Option::Call, strikePrice_));
    shared_ptr<Exercise> exercise(
        new EuropeanExercise(exerciseDate_));
    option_ = shared_ptr<VanillaOption>(
        new VanillaOption(payoff, exercise));
    marketValue_ = blackPrice(volatility->value());
}
Real modelValue() const {
    option_->setPricingEngine(engine_);
    return option_->NPV();
}
Real blackPrice(Real volatility) const {
    return blackFormula(Option::Call,
        // ...volatility, stored parameters...
    );
}
void addTimesTo(std::list<Time>&) const {}
private:
    shared_ptr<VanillaOption> option_;
    // other data members, not shown
};

```

---

Anyway: the implementation is not complex. The constructor takes information on the underlying contract such as maturity and strike, as well as the quoted volatility and other needed market data; it also allows one to choose how to calculate the calibration error. Some data are passed to the base class constructor, while some other are stored in data members; finally, the data are used to instantiate and store a `VanillaOption` instance and its market price is calculated. The option and its price are also stored in the corresponding data members.

It's not easy to spot it at first sight, but there's a small problem in the initialization of the instance. The time to maturity `tau_` is calculated in the constructor as the time between today's date and the exercise date. Unfortunately, this doesn't take into account the fact that today's date might change. In order to work correctly even in this case, this class should recalculate the values of

`tau_` and `marketValue_` each time they might have changed—that is, inside an override of the `performCalculations` method.

The rest is straightforward. As I mentioned before, the `modelValue` method is implemented by setting the model-based engine to the instrument and asking it for its NPV (this would work for any model, not just the Heston one, which explains my discontent at the class name). The `blackPrice` class returns the result of the Black-Scholes formula based on the value of the passed volatility and of the other stored parameters.<sup>5</sup> Finally, the `addTimesTo` method does nothing; this is the only Heston-specific part, since we don't have a tree-based Heston model. To be fully generic, we should return here the exercise time (that would be the `tau_` data member discussed above) so that it can be used by some other model.

That's all. The machinery of the base class will use these methods and provide the functionality that is needed for the calibration. But before going into that, I need to make a short detour.

### Aside: breaking assumptions

What happens if the assumptions baked in the code don't hold—for instance, because the volatility is not quoted by means of a Black model, or the instrument is not quoted in terms of its volatility at all?

In this case, the machinery still works but we're left with misnomers that will make it difficult to understand the code. For instance, if the formula used to pass from volatility to price doesn't come from a Black model, we still have to implement it in the `blackPrice` method, leading to much head-scratching; and if the instrument price is quoted instead of the volatility, we have to store it in the `volatility_` data member and to implement the `blackPrice` method, rather puzzlingly, as:

```
Real blackPrice(Real volatility) const {
    return volatility;
}
```

A more acceptable design might use more generic names (such as `quote` instead of `volatility`, or `marketPrice` instead of `blackPrice`) and leave it to derived classes to make their own shtick more clear—say, by renaming method arguments, since the compiler doesn't care. Then again, it might still be confusing for humans reading the code.

## 5.2 Parameters

There is an ambiguity when we say that a model has a given number of parameters. If they are constant, all is well; for instance, we can safely say that the Hull-White model has two parameters  $\alpha$  and  $\sigma$ . What if one of the two was time-dependent, though? In turn, it would have some kind of

---

<sup>5</sup>The price could also be obtained by setting a Black engine to the stored instrument and asking for its NPV. This is the route chosen by other helpers in the library.

parametric form. Conceptually, it would still be one single model parameter; but it might add several numbers to the set to be calibrated.

The `Parameter` class, shown in the listing below, takes the above into account—and, unfortunately, embraces the ambiguity: it uses the term “parameter” for both the instances of the class (that represent a model parameter, time-dependent or not) and for the numbers underlying their parametric forms. Our bad: you’ll have to be careful not to get confused in the discussion that follows.

Sketch of the `Parameter` class.

---

```

class Parameter {
    protected:
        class Impl {
            public:
                virtual ~Impl() {}
                virtual Real value(const Array& params, Time) const = 0;
            };
            shared_ptr<Impl> impl_;
        public:
            Parameter();
            const Array& params() const;
            void setParam(Size i, Real x) { params_[i] = x; }
            bool testParams(const Array& params) const;
            Size size() const { return params_.size(); }
            Real operator()(Time t) const {
                return impl_->value(params_, t);
            }
        protected:
            Parameter(Size size,
                      const shared_ptr<Impl>& impl,
                      const Constraint& constraint);
            Array params_;
            Constraint constraint_;
};
```

---

Specialized behavior for different parameter will be implemented in derived classes (I’ll show you a few of those shortly). However, the way we go about it is somewhat unusual: instead of declaring a virtual method directly, the `Parameter` class is given an inner class `Impl`, which declares a purely virtual `value` method. There’s method in this madness;<sup>6</sup> but let me gloss over it for now. The idiom is used in other classes, and will be explained in [appendix A](#).

---

<sup>6</sup>Someone on the Wilmott forums suggested that the reason is job security: if we obfuscate the code enough, nobody else will be able to maintain it. I can see his point, but this is not the reason.

Instances of `Parameter` represent, in principle, a time-dependent parameter and store an array `params_` which contain the values of the parameters used to describe its functional form. Most of the interface deals with these underlying parameters; the `params` method returns the whole array, the `setParam` method allows to change any of the values, and the `size` method returns their number.

`Parameter` instances also store a constraint that limits the range of values that the underlying parameters can take; the `testParam` method provides the means to check their current values against the constraint. Finally, `operator()` returns the value of the represented parameter<sup>7</sup> as a function of time, given the current values of the underlying parameters; as I mentioned, the actual implementation is delegated to the stored instance of the inner `Impl` class.

Finally, the class declared a couple of constructors. One is protected, and allows derived classes to initialize their own instances. Another is public; it creates instances without behavior (and therefore useless) but allows us to use `Parameter` with containers such as `std::vector`.<sup>8</sup>

The listing that follows shows a few examples of actual parameters; they inherit from the `Parameter` class and declare inner `Impl` classes that inherit from `Parameter::Impl` and implement the required behavior.

Sketch of a few classes inherited from `Parameter`.

---

```

class ConstantParameter : public Parameter {
    class Impl : public Parameter::Impl {
        public:
            Real value(const Array& params, Time) const {
                return params[0];
            }
        };
    public:
        ConstantParameter(const Constraint& constraint)
        : Parameter(1, /* ... */) {}
    };
}

class NullParameter : public Parameter {
    class Impl : public Parameter::Impl {
        public:
            Real value(const Array&, Time) const {
                return 0.0;
            }
        };
    public:
        NullParameter() : Parameter(0, /* ... */) {}
    };
}

```

---

<sup>7</sup>I'm not sure how I should call it. The main parameter? The outer parameter?

<sup>8</sup>I think this is no longer necessary in C++11, but we are still living in the past.

```

class PiecewiseConstantParameter : public Parameter {
    class Impl : public Parameter::Impl {
        public:
            Impl(const std::vector<Time>& times);
            Real value(const Array& params, Time t) const {
                for (Size i=0; i<times_.size(); i++) {
                    if (t<times_[i])
                        return params[i];
                }
                return params.back();
            }
        private:
            std::vector<Time> times_;
    };
    public:
        PiecewiseConstantParameter(const std::vector<Time>& times,
                                   const Constraint& constraint)
        : Parameter(times.size()+1, /* ... */){}
    };
}

```

---

The first represents a parameter which is constant in time; this is what we usually think about when we talk of a parameter, and is possibly the most used. The array of internal parameters has just one element (as seen by the 1 passed to the `Parameter` constructor), and that's what the implementation returns independently of the passed time.

The second is a null parameter; its value is supposed to be 0 and must not be calibrated. In this case, the stored array has no elements (again, see the 0 passed to the `Parameter` constructor) since nothing will move during calibration. This could probably be a specific case of a more general `FixedParameter` class, whose fixed value could be different from 0.

Finally, the third class represents a time-dependent parameter. It is modeled as piecewise constant between any two consecutive times in a given set; thus, if the set has  $n$  times, we'll have  $n+1$  different values (including the one before the first time and the one after the last). The implementation is a simple linear search, as we don't expect the times to be too many or we'd be likely to over-calibrate.

The library implements other parameter classes (and we could define others; for instance, using other parameterizations of time dependence) but I won't keep you any longer. At this point, we need to turn to the class that—I don't really have a less awkward way to say it—models a calibrated model.

## 5.3 The `CalibratedModel` class

The implementation of the `CalibratedModel` class is shown in the listing below.

---

**Implementation of the `CalibratedModel` class.**


---

```

class CalibratedModel : public virtual Observer,
                           public virtual Observable {
public:
    CalibratedModel(Size nArguments)
        : arguments_(nArguments),
          constraint_(new PrivateConstraint(arguments_)),
          shortRateEndCriteria_(EndCriteria::None) {}

    void update() {
        generateArguments();
        notifyObservers();
    }
    Disposable<Array> params() const;
    virtual void setParams(const Array& params);
    void calibrate(
        const vector<shared_ptr<CalibrationHelper> >&,
        OptimizationMethod& method,
        const EndCriteria& endCriteria,
        const Constraint& constraint = Constraint(),
        const vector<Real>& weights = vector<Real>());
    EndCriteria::Type endCriteria();
protected:
    virtual void generateArguments() {}
    vector<Parameter> arguments_;
    shared_ptr<Constraint> constraint_;
    EndCriteria::Type shortRateEndCriteria_;
private:
    class PrivateConstraint;
    class CalibrationFunction;
};

Disposable<Array> CalibratedModel::params() const {
    Size size = 0, i;
    for (i=0; i<arguments_.size(); i++)
        size += arguments_[i].size();
    Array params(size);
    Size k = 0;
    for (i=0; i<arguments_.size(); i++) {
        for (Size j=0; j<arguments_[i].size(); j++, k++) {
            params[k] = arguments_[i].params()[j];
        }
    }
}

```

```

return params;
}

void CalibratedModel::setParams(const Array& params) {
    Array::const_iterator p = params.begin();
    for (Size i=0; i<arguments_.size(); ++i) {
        for (Size j=0; j<arguments_[i].size(); ++j, ++p) {
            QL_REQUIRE(p!=params.end(), "too few parameters");
            arguments_[i].setParam(j, *p);
        }
    }
    QL_REQUIRE(p==params.end(), "too many parameters");
    generateArguments();
    notifyObservers();
}

void CalibratedModel::calibrate(
    const vector<shared_ptr<CalibrationHelper> >& instruments,
    OptimizationMethod& method,
    const EndCriteria& endCriteria,
    const Constraint& additionalConstraint,
    const vector<Real>& weights) {

    Constraint c =
        additionalConstraint.empty() ?
        *constraint_ :
        CompositeConstraint(*constraint_, additionalConstraint);
    CalibrationFunction f(this, instruments, weights);
    Problem prob(f, c, params());
    shortRateEndCriteria_ = method.minimize(prob, endCriteria);
    setParams(prob.currentValue());
    notifyObservers();
}

```

---

Its core is the `calibrate` method, with most other features being there in order to support its execution. (In fact, there's a couple of public methods there that should be used, directly or indirectly, by `calibrate` alone and should belong to the protected section. I'm not sure that, in true Ellery Queen tradition, you have all the clues you need; but you can try looking at the code and guessing which ones.)

Instances of this class store a vector of `Parameter` instances, which for some reason are called `arguments` here; a constraint, to be applied to the set of their underlying parameters; and a member of the `EndCriteria::Type` enumeration, which tells us how the latest calibration ended (say, because

it succeeded, or for reaching the maximum number of evaluations) and whose name still shows the original use of this class for short-rate models. As I mentioned, we gave this class little attention for quite a while.

The constructor takes a single argument specifying the number of model parameters and initializes the data members: the vector of parameters is given the passed size, the constraint is set to an instance of an inner `PrivateConstraint` class that I'll describe later, and the end criterion is set to `None` since the model is not yet calibrated.

Now, we have an interesting glitch here. This constructor is obviously meant to be used by derived classes, but is declared as public (probably an oversight). This, together with the fact that the class doesn't define any pure virtual function, makes it possible to create instances of `CalibratedModel` directly; however, such instances are unusable since they don't provide a way to set their parameters to anything useful (the stored `Parameter` instances are default-constructed and thus lack any behavior). Technically, fixing this glitch would break backward compatibility; but it might be argued that programs using this feature were broken anyway. We'll think about it in one of the next versions.

When the model receives a notification, the `update` method notifies the model's own observers after performing any needed calculation. These will be implemented by overriding the virtual `generateArguments` method. The name might be misleading, since it seems to suggest that parameter instances should be created here; but this can't be, since we don't want to override parameters that we might have already calibrated.<sup>9</sup> Instead, this method is used either to create parameters that don't need calibration (e.g., the term-structure parameter in some short-rate models, which follows the risk-free rate) or to perform some housekeeping, as we'll see in the continuation of the Heston model example.

The `params` and `setParams` methods are used to read and write the underlying parameters. To return them, the `params` method asks each of the stored `Parameter` instances for the number of underlying parameters it provides, creates an `Array` instance that can hold all of them, and collects their values (it can't add them to the array in a single loop because `Array` doesn't provide a `push_back` operation). In a similar way, the `setParams` method reads from the passed array and writes the required number of values in the stored `Parameter` instances.

Now, if you guessed that `params` and `setParams` are the methods that I'd rather have in the protected section, you can go and pour yourself a beverage of your choice.<sup>10</sup> These two methods should only be called from inside the `calibrate` method; client code isn't even able to know the number of the underlying parameters or which ones belong to each `Parameter` instance,<sup>11</sup> so it shouldn't be able to modify them, and it has very little use for reading them, too.

As I said, the `calibrate` method is the focus of the class; and in true managerial fashion, it delegates most of the work to other objects. Simply put, it sets up a minimization problem so that solving it gives the calibrated set of parameters. The ingredients of the problem are an instance of a class derived from `OptimizationMethod`<sup>12</sup> which is passed to the method by the calling code; a function

<sup>9</sup>We don't want to recalibrate at each notification, either.

<sup>10</sup>Drink responsibly. Also, don't drink and code.

<sup>11</sup>Unless its programmer reads the source code of the particular model used. That's cheating, though.

<sup>12</sup>For instance, it might implement the simplex method or Levenberg-Marquardt. More details on those are in appendix A.

to minimize, or rather a function object, which is an instance of its inner `CalibrationFunction` class and that returns a measure of the calibration error (more on that shortly); and a constraint on the parameter values, which defaults to the instance of `PrivateConstraint` stored at construction and can optionally be combined with an additional constraint passed as an argument.

The `calibrate` method collects all of the above, instantiates the problem, and starts the minimization. When that is done, it saves the end criterion, sets the parameter values to those that minimize the error (that is, those returned by `problem.currentValue()`) and notifies any observers that something has changed. The end criterion (which might be that the minimization succeeded, or that it failed for a number of different reasons) can be retrieved by means of the `endCriteria` method. If I were to write this class now, I'd probably return it from `calibrate` instead; but I'm ambivalent about it. It might make sense to store it in the model.

As it is now, the `calibrate` method provide little exception safety; if an exception were thrown at some point during the calibration, the model would be left with the last parameter values tried by the minimizer (the very same that probably caused the exception to be thrown). We should set an appropriate end criterion instead, so that it can be checked by the calling code: this is what happens if the calibration fails for other reasons.

The last pieces of functionality are implemented in the `PrivateConstraint` and `CalibrationFunction` inner classes, shown in the next listing.

Inner classes of the `CalibratedModel` class.

---

```

class CalibratedModel::PrivateConstraint : public Constraint {
private:
    class Impl : public Constraint::Impl {
        const vector<Parameter>& arguments_;
    public:
        Impl(const vector<Parameter>& arguments);
        bool test(const Array& params) const {
            for (Size i=0; i<arguments_.size(); i++) {
                Array testParams(/* `select the correct subset` */);
                if (!arguments_[i].testParams(testParams))
                    return false;
            }
            return true;
        }
    };
public:
    PrivateConstraint(const vector<Parameter>& arguments);
};

class CalibratedModel::CalibrationFunction
    : public CostFunction {
public:

```

```

CalibrationFunction(
    CalibratedModel* model,
    const vector<shared_ptr<CalibrationHelper>>& instruments,
    const vector<Real>& weights)
: model_(model, no_deletion), instruments_(instruments),
weights_(weights) {}

virtual Disposable<Array> values(const Array& params) const {
    model_->setParams(params);
    Array values(instruments_.size());
    for (Size i=0; i<instruments_.size(); i++) {
        values[i] = instruments_[i]->calibrationError()
            *sqrt(weights_[i]);
    }
    return values;
}
virtual Real value(const Array& params) const;
private:
    shared_ptr<CalibratedModel> model_;
    const vector<shared_ptr<CalibrationHelper>>& instruments_;
    vector<Real> weights_;
};

```

---

The `PrivateConstraint` class works by collecting the constraints set to the stored `Parameter` instances. The logic is similar to that of `params` or `setParams`; it loops over the stored parameters, determines the subset of underlying parameters that belong to each one, and asks the `Parameter` instance to check them by calling its `testParams` method. The composite constraint is satisfied if and only if all the inner constraints are.

Finally, the `CalibrationFunction` class provides an estimate of the calibration error for a given set of parameters. Its constructor takes and stores a pointer to the model being calibrated, the set of quoted instruments being used for the calibration, and a set of weights. For some reason, the pointer to the model is stored in a `shared_ptr` instance, taking care that it's not deleted with the calibration function. Storing the raw pointer would have been enough.

The class inherits from `CostFunction`, which requires derived classes to implement both a `values` method returning a set of errors (in this case, one per quoted instrument) and a `value` method returning a single error estimate; a given optimization method might use the one or the other. The two work in a similar way, so I'm showing the implementation of just the first in the listing: they set the given parameters to the model, and then ask the stored helpers for the calibration error. For this to work, the helpers need to be set a pricing engine that uses the model being calibrated. The setup of the entire thing would be something like this:

```

shared_ptr<HestonModel> model(...);
vector<shared_ptr<CalibrationHelper>> helpers(...);
shared_ptr<PricingEngine> engine =
    make_shared<AnalyticHestonEngine>(model, ...);
for (i=0; i<helpers.size(); ++i)
    helpers[i]->setPricingEngine(engine);
model->calibrate(helpers, ...);

```

That is, the helpers use the engine to calculate their model prices, and in turn the engine uses the model. Thus, after the call to `setParams` inside the `values` method the model prices change and so do the corresponding calibration errors. The `values` method returns the set of distinct errors, while the `value` method returns the sum of their squares.

A final note: currently, the `CalibrationFunction` class is declared as a friend of `CalibratedModel`. This is actually not necessary, since it only accesses the public `setParams` method; and if we were using C++11, it wouldn't be necessary even if `setParams` was protected. According to the new standard, inner classes have access to all member of their enclosing class, public or not.

### 5.3.1 Example: the Heston model, continued

Time for the second part of the example. The code for the `HestonModel` class is shown in the following listing.

Implementation of the `HestonModel` class.

---

```

class HestonModel : public CalibratedModel {
    public:
        HestonModel(const shared_ptr<HestonProcess>& process)
        : CalibratedModel(5), process_(process) {
            arguments_[0] = ConstantParameter(process->theta(),
                                              PositiveConstraint());
            arguments_[1] = ConstantParameter(process->kappa(),
                                              PositiveConstraint());
            arguments_[2] = ConstantParameter(process->sigma(),
                                              PositiveConstraint());
            arguments_[3] =
                ConstantParameter(process->rho(),
                                  BoundaryConstraint(-1.0, 1.0));
            arguments_[4] = ConstantParameter(process->v0(),
                                              PositiveConstraint());
        generateArguments();
        registerWith(process_->riskFreeRate());
        registerWith(process_->dividendYield());
        registerWith(process_->s0());

```

```

    }

    Real theta() const { return arguments_[0](0.0); }
    Real kappa() const { return arguments_[1](0.0); }
    Real sigma() const { return arguments_[2](0.0); }
    Real rho() const { return arguments_[3](0.0); }
    Real v0() const { return arguments_[4](0.0); }

    shared_ptr<HestonProcess> process() const {
        return process_;
    }

protected:
    void generateArguments() {
        process_.reset(
            new HestonProcess(process_>riskFreeRate(),
                process_>dividendYield(),
                process_>s0(),
                v0(), kappa(), theta(),
                sigma(), rho()));
    }

    shared_ptr<HestonProcess> process_;
};


```

---

As you might know, the model has five parameters  $\theta$ ,  $\kappa$ ,  $\sigma$ ,  $\rho$  and  $v_0$ . The process it describes for its underlying also depends on the risk-free rate, its current value, and possibly a dividend yield. For reasons that will become more clear in [chapter 6](#), the library groups all of those in a separate `HestonProcess` class (for brevity, I'm not showing its interface here; we're just using it as the container for the model parameters).

The `HestonModel` constructor takes an instance of the process class, stores it, and defines the parameters to calibrate. First it passes their number (5) to its base class constructor, then it builds each of them. They are all constant parameters;  $\rho$  is constrained to be between  $-1$  and  $1$ , while the others must all be positive. Their initial values are taken from the process. The class defines the inspectors `theta`, `kappa`, `sigma`, `rho` and `v0` to retrieve their current values; each of them returns the value of the corresponding `Parameter` instance at time  $t = 0$  (which is as good as any other time, since the parameters are constant).

After the parameters are built, the `generateArguments` method is called. This will also be called each time the parameters change during calibration, and replaces the stored process instance with another one containing the same term structures and quote as the old one but with the new parameters. The reason for this is that the new process would be ready if any engine were to require it from the model; but I wonder if the `process` inspector should not build the process on demand instead. If the actual process is not required except as a holder of parameters and curves, we could define inspectors for all of them instead, have the engine use them directly, and save ourselves the

creation of a new complex object at each step of the calibration. You're welcome to do the experiment and time the new implementation against the current one.

Finally, the constructor registers with the relevant observables. The process instance will be replaced by `generateArguments`, so there's no point in registering with it. Instead, we register directly with the contained handles, that will be moved inside each new process instance.

Together with the inspectors I already mentioned, this completes the implementation of the model. The calibration machinery is inherited from the `CalibratedModel` class, and the only thing that's needed to make it work is an engine that takes a `HestonModel` instance and uses it to price the `VanillaOption` instances contained in the calibration helpers.

You'll forgive me for not showing here the `AnalyticHestonEngine` class provided by QuantLib: it implements a closed formula for European options under the Heston model, cites as many as five papers and books in its comments, and goes on for about 650 lines of code. For the non mathematically-minded, it is *Cthulhu fhtagn* stuff. If you're interested, the full code is available in the library.

# 6. The Monte Carlo framework

The concept behind Monte Carlo methods is deceptively simple—generate random paths, price the desired instrument over such paths, and average the results. However (and even assuming that the tasks I just enumerated were as simple as they seem: they’re not) several choices can be made regarding the implementation of the model. Pseudo-random or quasi-random numbers? Which ones? When should the simulation stop? What results should be returned? How about more advanced techniques such as likelihood ratio or pathwise Greeks?

The mission of library writers, should they accept it, is to make most (possibly all) such choices independently available to the user; to make it possible to model the underlying stochastic process in a polymorphic way; and on top of it all, to provide at least part of the scaffolding needed to put everything together and run the simulation.

The first part of this chapter describes the design of our Monte Carlo framework and in what measure it meets (or falls short of) the above requirements; the second part outlines how it can be integrated with the pricing-engine framework described in [chapter 2](#).

## 6.1 Path generation

The task of generating random paths puts together a number of building blocks and concepts; in our implementation, it also puts together a number of programming techniques. In this section, I’ll describe such blocks and how they ultimately yield a path-generator class.

### 6.1.1 Random-number generation

Among the basic building blocks for our task are random-number generators (RNG), the most basic being uniform RNGs. The library provides quite a few of them, the most notable being the Mersenne Twister among pseudo-random generators and Sobol among low-discrepancy ones. Here, I’ll gloss over their implementation, as well as over when and how you should use either kind of generator; such information (as well as everything else related to Monte Carlo methods) is covered in literature much better than I ever could, for instance in [Glasserman, 2003](#) or [Jäckel, 2002](#).

The interface of uniform RNGs, shown in the listing below, doesn’t make for a very exciting reading.

Basic random-number generator classes.

---

```

template <class T>
struct Sample {
    typedef T value_type;
    T value;
    Real weight;
};

class MersenneTwisterUniformRng {
    public:
        typedef Sample<Real> sample_type;
        explicit MersenneTwisterUniformRng(unsigned long seed = 0);
        sample_type next() const;
};

class SobolRsg {
    public:
        typedef Sample<std::vector<Real> > sample_type;
        SobolRsg(Size dimensionality,
                 unsigned long seed = 0);
        const sample_type& nextSequence() const;
        Size dimension() const { return dimensionality_; }
};

```

---

Its methods are the ones you would expect: pseudo-random generators sport a constructor that takes an initialization seed and a method, `next`, that returns the next random value; low-discrepancy generators take an additional constructor argument that specify their dimensionality and return a vector of values from their `nextSequence` method.<sup>1</sup> The only feature of note is the `Sample` struct, used to return the generated values, which also contains a weight. This was done to make it possible for generators to sample values with some kind of bias (e.g., when using importance sampling). However, at this time the library doesn't contain any such generator; all available RNGs return samples with unit weight.

So much for uniform RNGs. However, they're only the rawest material; we need to refine them further. This is done by means of classes (wrappers, or combinatorics, if you will; or again, you might see them as implementations of the Decorator pattern) that take an instance of a uniform RNG and yield a different generator. We need two kinds of decorators. On the one hand, we need tuples of random numbers rather than single ones: namely,  $M \times N$  numbers in order to evolve  $M$  variables for  $N$  steps. Low-discrepancy generators return tuples already (of course they need to do so, for their low-discrepancy property to hold at the required dimensionality). Since we want a generic interface,

<sup>1</sup>On second thought, we might have called the `nextSequence` method simply `next`. Including the “sequence” token in the method name reminds me of Hungarian notation, which is up there with `goto` in the list of things considered harmful.

we'll make the pseudo-random generators return tuples as well by drawing from them repeatedly; the class template that does so, called `RandomSequenceGenerator`, is shown in the next listing.

Sketch of the `RandomSequenceGenerator` class template.

---

```
template<class RNG>
class RandomSequenceGenerator {
public:
    typedef Sample<std::vector<Real>> sample_type;
    RandomSequenceGenerator(Size dimensionality,
                           const RNG& rng)
        : dimensionality_(dimensionality), rng_(rng),
          sequence_(std::vector<Real>(dimensionality), 1.0) {}
    const sample_type& nextSequence() const {
        sequence_.weight = 1.0;
        for (Size i=0; i<dimensionality_; i++) {
            typename RNG::sample_type x(rng_.next());
            sequence_.value[i] = x.value;
            sequence_.weight *= x.weight;
        }
        return sequence_;
    }
private:
    Size dimensionality_;
    RNG rng_;
    mutable sample_type sequence_;
};
```

---

On the other hand, we usually want Gaussian random numbers, not uniform ones. There's a number of ways to obtain them, but not all of them might be equally suited for any given situation; for instance, those consuming  $M$  uniform random numbers to yield  $N$  Gaussian ones don't work with low-discrepancy generators (especially if  $M$  varies from draw to draw, like in rejection techniques). If you want to stay generic, the simplest method is probably to feed your uniform numbers to the inverse-cumulative Gaussian function, which yield a Gaussian per uniform; this is also the default choice in the library. The corresponding class template, `InverseCumulativeRsg`, is shown in the listing below. It takes the implementation of the inverse-cumulative function as a template argument, since there's a few approximations and no closed formula for it; most of the times you'll be fine with the one provided by the library, but at times one might choose to change it in order to trade accuracy for speed (or the other way around).

Sketch of the `InverseCumulativeRsg` class template.

---

```
template <class USG, class IC>
class InverseCumulativeRsg {
public:
    typedef Sample<std::vector<Real> > sample_type;
    InverseCumulativeRsg(const USG& uniformSequenceGenerator,
                         const IC& inverseCumulative);
    const sample_type& nextSequence() const {
        typename USG::sample_type sample =
            uniformSequenceGenerator_.nextSequence();
        x_.weight = sample.weight;
        for (Size i = 0; i < dimension_; i++) {
            x_.value[i] = ICD_(sample.value[i]);
        }
        return x_;
    }
private:
    USG uniformSequenceGenerator_;
    Size dimension_;
    mutable sample_type x_;
    IC ICD_;
};
```

---

At this point, you might have started to have bad feelings about this framework. I have only covered random numbers, and yet we already have two of three choices to make in order to instantiate a generator. By the time we're done, we might have a dozen of template arguments on our hands. Well, bear with me for a few more pages. In a later section, I'll describe how the library tries to mitigate the problem.

A final note: as you might have noticed, the sequence generators shown in the listings keep a `mutable` sequence as a data member and fill it with the new draws in order to avoid copies. Of course, this prevents instances of such classes to be shared by different threads—although I'd be a very happy camper if all the problems QuantLib had with multithreading were like this one.

### Aside: the road more traveled.

The 2011 C++ standard includes RNGs as part of its standard library. At some point, we'll probably switch to their interface; but I still don't know when nor how we'll do it. We might ditch our implementation and switch to the standard classes; or we could replace the interface of our classes while keeping their current implementation; or, again, we could provide adaptors between the

two interfaces. In any case, the changes would break backward compatibility, which means going through deprecation of the old ones for a few releases and their coexistence with the new ones.

## 6.1.2 Stochastic processes

The whole point of using RNGs is, of course, to use the generated random numbers to drive a stochastic process. The interface of the `StochasticProcess` class (displayed in the listing that follows) was designed with this application in mind—and it shows.

Partial implementation of the **StochasticProcess** class.

```

virtual Disposable<Matrix> covariance(Time t0,
                                         const Array& x0,
                                         Time dt) const;
virtual Disposable<Array> evolve(Time t0,
                                    const Array& x0,
                                    Time dt,
                                    const Array& dw) const;
virtual Disposable<Array> apply(const Array& x0,
                                 const Array& dx) const;
};

Size StochasticProcess::factors() const {
    return size();
}

Disposable<Array>
StochasticProcess::expectation(Time t0,
                               const Array& x0,
                               Time dt) const {
    return apply(x0, discretization_->drift(*this,t0,x0,dt));
}

Disposable<Matrix>
StochasticProcess::stdDeviation(Time t0,
                                const Array& x0,
                                Time dt) const {
    return discretization_->diffusion(*this,t0,x0,dt);
}

Disposable<Matrix>
StochasticProcess::covariance(Time t0,
                              const Array& x0,
                              Time dt) const {
    return discretization_->covariance(*this,t0,x0,dt);
}

Disposable<Array>
StochasticProcess::evolve(Time t0, const Array& x0,
                          Time dt, const Array& dw) const {
    return apply(expectation(t0,x0,dt),
                  stdDeviation(t0,x0,dt)*dw);
}

```

```
Disposable<Array>
StochasticProcess::apply(const Array& x0,
                        const Array& dx) const {
    return x0 + dx;
}
```

---

The concept modeled by the class is a somewhat generic n-dimensional stochastic process described by the equation

$$dx = \mu(t, x)dt + \sigma(t, x) \cdot dw$$

(yes, it took six chapters, but there's an equation in this book. I was a physicist once, you know). However, the interface of the class deals more with the discretization of the process over the time steps of a sampled path.

Straight away, the class defines a polymorphic inner class `discretization`. It is an instance of the Strategy pattern, whose purpose is to make it possible to change the way a process is discretely sampled. Its methods take a `StochasticProcess` instance, a starting point  $(t, x)$ , and a time step  $\Delta t$  and returns a discretization of the process drift and diffusion.<sup>2</sup>

Back to the `StochasticProcess` interface. The class defines a few inspectors. The `size` method returns the number of variables modeled by the process. For instance, it would return 1 for a regular Black-Scholes process with a single underlying, or 2 for a stochastic-volatility model, or  $N$  for the joint process of  $N$  correlating underlying assets each following a Black-Scholes process. The `factors` method returns the number of random variates used to drive the variables; it has a default implementation returning the same number as `size`, although on afterthought I'm not sure that it was a great idea; we might have required to specify that explicitly. To round up the inspectors, the `initialValues` method returns the values of the underlying variables at  $t = 0$ ; in other words, the present values, in which no randomness is involved. As in most `StochasticProcess` methods, the result is wrapped in a `Disposable` class template. This is a way to avoid a few copies when returning arrays or matrices; if you're interested in the details, you can have a look at appendix A. When we can rely on compilers supporting the C++11 standard, we'll be able to replace those with r-value references ([Hinnant et al, 2006](#)); I'm currently planning to drop support for C++03 in early 2021 (as usual, dropping support for older compilers will involve a few releases in which we deprecate them, so that we can finally do the jump without leaving users outside in the rain).

The next pair of methods (also inspectors, in a way) return more detailed information on the process. The `drift` method return the  $\mu(t, x)$  term in the previous formula, while the `diffusion` method returns the  $\sigma(t, x)$  term. Obviously, for dimensions to match, the drift term must be a  $N$ -dimensional array and the diffusion term a  $N \times M$  matrix, with  $M$  and  $N$  being the dimensions returned by the `factors` and `size` methods, respectively. Also, the diffusion matrix must include correlation information.

<sup>2</sup>by this time, you'll have surely noticed that there's no provision for jumps. Yes, I know. Hold that thought; I'll have more to say on this later.

The last set of methods deals with the discretization of the process over a path. The `expectation` method returns the expectation values of the process variables at time  $t + \Delta t$ , assuming they had values  $\mathbf{x}$  at time  $t$ . Its default implementation asks the stored `discretization` instance for the integrated drift of the variables over  $\Delta t$  and applies it to the starting values (`apply` is another virtual method that takes a value  $\mathbf{x}$  and a variation  $\Delta\mathbf{x}$  and unsurprisingly returns  $\mathbf{x} + \Delta\mathbf{x}$ . Why we needed a polymorphic method to do this, you ask? It's a sad story that will be told in a short while). The `stdDeviation` method returns the diffusion term integrated over  $t + \Delta t$  and starting at  $(t, \mathbf{x})$ , while the `covariance` method returns its square; in their default implementation, they both delegate the calculation to the `discretization` instance. Finally, the `evolve` method provides some kind of higher-level interface; it takes a starting point  $(t, \mathbf{x})$ , a time step  $\Delta t$ , and an array  $\mathbf{w}$  of random Gaussian variates, and returns the end point  $\mathbf{x}'$ . Its default implementation asks for the standard-deviation matrix, multiplies it by the random variates to yield the random part of the simulated path, and applies the resulting variations to the expectation values of the variables at  $t + \Delta t$  (which includes the deterministic part of the step).

As you've seen, the methods in this last set are all given a default implementation, sometimes calling methods like `drift` or `diffusion` by way of the discretization strategy and sometimes calling directly other methods in the same set; it's a kind of multiple-level Template Method pattern, as it were. This makes it possible for classes inheriting from `StochasticProcess` to specify their behavior at different levels.

At the very least, such a class must implement the `drift` and `diffusion` methods, which are purely virtual in the base class. This specifies the process at the innermost level, and is enough to have a working stochastic process; the other methods would use their default implementation and delegate the needed calculations to the contained `discretization` instance. The derived class can prescribe a particular discretization, suggest one by setting it as a default, or leave the choice entirely to the user.

At a somewhat outer level, the process class can also override the `expectation` and `stdDeviation` methods; this might be done, for instance, when it's possible to write an closed formula for their results. In this case, the developer of the class can decide either to prevent the constructor from taking a `discretization` instance, thus forcing the use of the formula; or to allow the constructor to take one, in which case the overriding method should check for its presence and possibly forward to the base-class implementation instead of using its own. The tasks of factoring in the random variates and of composing the integrated drift and diffusion terms are still left to the `evolve` method.

At the outermost level, the derived class can override the `evolve` method and do whatever it needs—even disregard the other methods in the `StochasticProcess` interface and rely instead on ones declared in the derived class itself. This can serve as a Hail Mary pass for those process which don't fit the interface as currently defined. For instance, a developer wanting to add stochastic jumps to a process might add them into `evolve`, provided that a random-sequence generator can generate the right mixture of Gaussian and Poisson variates to pass to the method.

Wouldn't it be better to support jumps in the interface? Well, yes, in the sense that it would be better if a developer could work *with* the interface rather than *around* it. However, there are two kinds of problems that I see about adding jumps to the `StochasticProcess` class at this time.

On the one hand, I'm not sure what the right interface should be. For instance, can we settle on a single generalized formula including jumps? I wouldn't want to release an interface just to find out later that it's not the correct one; which is likely, given that at this time we don't have enough code to generalize from.

On the other hand, what happens when we try to integrate the next feature? Do we add it to the interface, too? Or should we try instead to generalize by not specializing; that is, by declaring fewer methods in the base-class interface, instead of more? For instance, we could keep `evolve` or something like it, while moving `drift`, `diffusion` and their likes in subclasses instead (by the way, this might be a way to work on a jump interface without committing too early to it: writing some experimental classes that define the new methods and override `evolve` to call them. Trying to make them work should provide some useful feedback).

But I went on enough already. In short: yes, Virginia, the lack of jumps in the interface is a bad thing. However, I think we'd be likely to make a poor job if we were to add them now. I, for one, am going to stay at the window and see if anybody comes along with some working code. If you wanted to contribute, those experimental classes would be most appreciated.

One last detour before we tackle a few examples. When implementing a one-dimensional process, the `Arrays` and their likes tend to get in the way; they make the code more tiresome to write and less clear to read. Ideally, we'd want an interface like the one declared by `StochasticProcess` but with methods that take and return simple real numbers instead of arrays and matrices. However, we can't declare such methods as overloads in `StochasticProcess`, since they don't apply to a generic process; and neither we wanted to have a separate hierarchy for 1-D processes, since they belong with all the others.

The library solves his problem by providing the `StochasticProcess1D` class, shown in the next listing.

Partial implementation of the **StochasticProcess1D** class.

```

    }
    virtual Real stdDeviation(Time t0, Real x0, Time dt) const {
        return discretization_->diffusion(*this, t0, x0, dt);
    }
    virtual Real variance(Time t0, Real x0, Time dt) const;
    virtual Real evolve(Time t0, Real x0,
                        Time dt, Real dw) const {
        return apply(expectation(t0,x0,dt),
                    stdDeviation(t0,x0,dt)*dw);
    }
private:
    Size size() const { return 1; }
    Disposable<Array> initialValues() const {
        return Array(1, x0());
    }
    Disposable<Array> drift(Time t, const Array& x) const {
        return Array(1, drift(t, x[0]));
    }
    // ...the rest of the StochasticProcess interface...
    Disposable<Array> evolve(Time t0, const Array& x0,
                             Time dt, const Array& dw) const {
        return Array(1, evolve(t0,x0[0],dt,dw[0]));
    }
};


```

---

On the one hand, this class declares an interface that mirrors exactly the `StochasticProcess` interface, even down to the default implementations, but uses the desired `Real` type. On the other hand, it inherits from `StochasticProcess` (establishing that a 1-D process “is-a” stochastic process, as we wanted) and implements its interface in terms of the new one by boxing and unboxing `Reals` into and from `Arrays`, so that the developer of a 1-D process needs not care about it: in short, a straightforward application of the Adapter pattern. Because of the 1:1 correspondence between scalar and vectorial methods (as can be seen in the listing) the 1-dimensional process will work the same way when accessed from either interface; also, it will have the same possibilities of customization provided by the `StochasticProcess` class and described earlier in this section.

\* \* \*

And now, an example or three. Not surprisingly, the first is the one-dimensional Black-Scholes process. You’ve already met briefly the corresponding class in [chapter 2](#), where it was passed (by the unwieldy name of `GeneralizedBlackScholesProcess`) as an argument to a pricing engine for a European option; I’ll go back to that code later on, when I discuss a few shortcomings of the

current design and possible future solutions. For the time being, let's have a look at the current implementation, sketched in the listing below.

Partial implementation of the `GeneralizedBlackScholesProcess` class.

---

```

class GeneralizedBlackScholesProcess
    : public StochasticProcess1D {
public:
    GeneralizedBlackScholesProcess(
        const Handle<Quote>& x0,
        const Handle<YieldTermStructure>& dividendTS,
        const Handle<YieldTermStructure>& riskFreeTS,
        const Handle<BlackVolTermStructure>& blackVolTS,
        const shared_ptr<discretization>& d =
            shared_ptr<discretization>());
    Real x0() const {
        return x0_->value();
    }
    Real drift(Time t, Real x) const {
        Real sigma = diffusion(t,x);
        Time t1 = t + 0.0001;
        return riskFreeRate_->forwardRate(t,t1,Continuous,...)
            - dividendYield_->forwardRate(t,t1,Continuous,...)
            - 0.5 * sigma * sigma;
    }
    Real diffusion(Time t, Real x) const;
    Real apply(Real x0, Real dx) const {
        return x0 * std::exp(dx);
    }
    Real expectation(Time t0, Real x0, Time dt) const {
        QL_FAIL("not implemented");
    }
    Real evolve(Time t0, Real x0, Time dt, Real dw) const {
        return apply(x0, discretization_->drift(*this,t0,x0,dt) +
            stdDeviation(t0,x0,dt)*dw);
    }
    const Handle<Quote>& stateVariable() const;
    const Handle<YieldTermStructure>& dividendYield() const;
    const Handle<YieldTermStructure>& riskFreeRate() const;
    const Handle<BlackVolTermStructure>& blackVolatility() const;
    const Handle<LocalVolTermStructure>& localVolatility() const;
};
```

---

Well, first of all, I should probably explain the length of the class name. The “generalized” bit doesn't

refer to some extension of the model, but to the fact that this class was meant to cover different specific processes (such as the Black-Scholes process proper, the Black-Scholes-Merton process, the Black process and so on); the shorter names were reserved for the specializations.<sup>3</sup>

The constructor of this class takes handles to the full set of arguments needed for the generic formulation: a term structure for the risk-free rate, a quote for the varying market value of the underlying, another term structure for its dividend yield, and a term structure for its implied Black volatility. In order to implement the `StochasticProcess` interface, the constructor also takes a `discretization` instance. All finance-related inputs are stored in the constructed class instance and can be retrieved by means of inspectors.

You might have noticed that the constructor takes a Black volatility  $\sigma_B(t, k)$ , which is not what is needed to return the  $\sigma(t, x)$  diffusion term. The process performs the required conversion internally,<sup>4</sup> using different algorithms depending on the type of the passed Black volatility (constant, depending on time only, or with a smile). The type is checked at run-time with a dynamic cast, which is not pretty but in this case is simpler than a Visitor pattern. At this time, the conversion algorithms are built into the process and can't be changed by the user; nor it is possible to provide a precomputed local volatility.

The next methods, `drift` and `diffusion`, show the crux of this class: it's actually a process for the logarithm of the underlying that masquerades as a process for the underlying itself. Accordingly, the term returned by `drift` is the one for the logarithm, namely,  $r(t) - q(t) - \frac{1}{2}\sigma^2(t, x)$ , where the rates and the local volatility are retrieved from the corresponding term structures (this is another problem in itself, to which I'll return). The `diffusion` method returns the local volatility.

The choice of  $\log(x)$  as the actual stochastic variable brings quite a few consequences. First of all, the variable we're using is not the one we're exposing. The quote we pass to the constructor holds the market value of the underlying; the same value is returned by the `x0` method; and the values passed to and returned from other methods such as `evolve` are for the same quantity. In retrospect, this was a bad idea (yes, I can hear you say "D'oh.") We were led to it by the design we had developed, but it should have been a hint that some piece was missing.

Other consequences can be seen all over the listing. The choice of  $\log(x)$  as the working variable is the reason why the `apply` method is virtual, and indeed why it exists at all: applying to  $x$  a drift  $\Delta$  meant for  $\log(x)$  is not done by adding them, but by returning  $x \exp(\Delta)$ . The `expectation` method is affected, too: its default implementation would apply the drift as above, thus returning  $\exp(E[\log(x)])$  instead of  $E[x]$ . To prevent it from returning the wrong value, the method is currently disabled (it raises an exception). In turn, the `evolve` method, whose implementation relied on `expectation`, had to be overridden to work around it.<sup>5</sup>

On top of the above smells, we have a performance problem—which is known to all those that tried a Monte Carlo engine from the library, as well as to all the people on the Wilmott forums to whom

<sup>3</sup>We might have used the Black-Scholes-Merton process as the most generic and inherited the others from it; but it doesn't feel right, given the "is-a" nature of inheritance.

<sup>4</sup>The class uses the Observer pattern to determine when to convert the Black volatility into the local one. Recalculation is lazy; meaning that it happens only when the local volatility is actually used and not immediately upon a notification from an observable.

<sup>5</sup>The `evolve` method might have been overridden for efficiency, even if `expectation` were available. The default implementation in `StochasticProcess1D` would return  $x \exp(\mu dt) \exp(\sigma dw)$ , while the overridden one calculates  $x \exp(\mu dt + \sigma dw)$ .

the results were colorfully described. Using `apply`, the `evolve` method performs an exponentiation at each step; but above all, the `drift` and `diffusion` methods repeatedly ask term structures for values. If you remember [chapter 3](#), this means going through at least a couple of levels of virtual method calls, sometimes (if we're particularly unlucky) retrieving a rate from discount factors that were obtained from the same rate to begin with.

Finally, there's another design problem. As I mentioned previously, we already used the Black-Scholes process in the `AnalyticEuropeanEngine` shown as an example in [chapter 2](#). However, if you look at the engine code in the library, you'll see that it doesn't use the process by means of the `StochasticProcess` interface; it just uses it as a bag of quotes and term structures. This suggests that the process is a jack of too many trades.

How can we fix it, then? We'll probably need a redesign. It's all hypothetical, of course; there's no actual code yet, just a few thoughts I had while I was reviewing the current code for this chapter. But it could go as follows.

First of all (and at the risk of being told once again that I'm overengineering) I'd separate the stochastic-process part of the class from the bag-of-term-structures part. For the second one, I'd create a new class, probably called something like `BlackScholesModel`. Instances of this class would be the ones passed to pricing engines, and would contain the full set of quotes and term structures; while we're at it, the interface should also allow the user to provide a local volatility, or to specify new ways to convert the Black volatility with some kind of Strategy pattern.

From the model, we could build processes. Depending on the level of coupling and the number of classes we prefer, we might write factory classes that take a model and return processes; we might add factory methods to the model class that return processes; or we might have the process constructors take a model. In any case, this could allow us to write process implementations that could be optimized for the given simulation. For instance, the factory (whatever it might be) might take as input the time nodes of the simulation and return a process that precomputed the rates  $r(t_i)$  and  $q(t_i)$  at those times, since they don't depend on the underlying value; if a type check told the factory that the volatility doesn't have smile, the process could precompute the  $\sigma(t_i)$ , too.

As for the  $x$  vs  $\log(x)$  problem, I'd probably rewrite the process so that it's explicitly a process for the logarithm of the underlying: the `x0` method would return a logarithm, and so would the other methods such as `expectation` or `evolve`. The process would gain in consistency and clarity; however, since it would be no longer guaranteed that a process generates paths for the underlying, client code taking a generic process would also need a function or a function object that converts the value of the process variable into a value of the underlying. Such function might be added to the `StochasticProcess` interface.

\* \* \*

Back to some existing code. For an example of a well-behaved process, you can look at the listing below, which shows the `OrnsteinUhlenbeckProcess` class.

Implementation of the `OrnsteinUhlenbeckProcess` class.

---

```

class OrnsteinUhlenbeckProcess : public StochasticProcess1D {
public:
    OrnsteinUhlenbeckProcess(Real speed,
                            Volatility vol,
                            Real x0 = 0.0,
                            Real level = 0.0);
    Real x0() const;
    ... // other inspectors
    Real drift(Time, Real x) const {
        return speed_ * (level_ - x);
    }
    Real diffusion(Time, Real) const {
        return volatility_;
    }
    Real expectation(Time t0, Real x0, Time dt) const {
        return level_ + (x0 - level_) * std::exp(-speed_*dt);
    }
    Real stdDeviation(Time t0, Real x0, Time dt) const {
        return std::sqrt(variance(t,x0,dt));
    }
    Real variance(Time t0, Real x0, Time dt) const {
        if (speed_ < std::sqrt(QL_EPSILON)) {
            return volatility_*volatility_*dt;
        } else {
            return 0.5*volatility_*volatility_/speed_*
                (1.0 - std::exp(-2.0*speed_*dt));
        }
    }
private:
    Real x0_, speed_, level_;
    Volatility volatility_;
};

```

---

The Ornstein-Uhlenbeck process is a simple one, whose feature of interest here is that its mean-reverting drift term  $\theta(\mu - x)$  and its constant diffusion term  $\sigma$  can be integrated exactly. Therefore, besides the mandatory `drift` and `diffusion` methods, the class also overrides the `expectation` and `stdDeviation` methods so that they implement the formulas for their exact results. The `variance` method (in terms of which `stdDeviation` is implemented) has two branches in order to prevent numerical instabilities; for small  $\theta$ , the formula for the variance is replaced by its limit

for  $\theta \rightarrow 0$ .

\* \* \*

Finally, for an example of a multi-dimensional process, we'll have a look at the `StochasticProcessArray` class, sketched in the next listing.

Partial implementation of the `StochasticProcessArray` class.

---

```

class StochasticProcessArray : public StochasticProcess {
    public:
        StochasticProcessArray(
            const std::vector<shared_ptr<StochasticProcess1D> >& ps,
            const Matrix& correlation)
        : processes_(ps), sqrtCorrelation_(pseudoSqrt(correlation)) {
            for (Size i=0; i<processes_.size(); i++)
                registerWith(processes_[i]);
        }
        // ...
        Disposable<Array> drift(Time t, const Array& x) const {
            Array tmp(size());
            for (Size i=0; i<size(); ++i)
                tmp[i] = processes_[i]->drift(t, x[i]);
            return tmp;
        }
        Disposable<Matrix> diffusion(Time t, const Array& x) const {
            Matrix tmp = sqrtCorrelation_;
            for (Size i=0; i<size(); ++i) {
                Real sigma = processes_[i]->diffusion(t, x[i]);
                std::transform(tmp.row_begin(i), tmp.row_end(i),
                              tmp.row_begin(i),
                              bind2nd(multiplies<Real>(), sigma));
            }
            return tmp;
        }
        Disposable<Array> expectation(Time t0, const Array& x0,
                                       Time dt) const;
        Disposable<Matrix> stdDeviation(Time t0, const Array& x0,
                                         Time dt) const;
        Disposable<Array> evolve(Time t0, const Array& x0,
                                  Time dt, const Array& dw) const {
            const Array dz = sqrtCorrelation_ * dw;
        }
}
```

```

    Array tmp(size());
    for (Size i=0; i<size(); ++i)
        tmp[i] = processes_[i]->evolve(t0, x0[i], dt, dz[i]);
    return tmp;
}
private:
    std::vector<shared_ptr<StochasticProcess1D>> processes_;
    Matrix sqrtCorrelation_;
};


```

---

It doesn't model a specific process, but rather the composition in a single entity of  $N$  correlated one-dimensional processes. I'll use it here to show how correlation information can be included in a process.

Its constructor takes the vector of 1-D processes for the underlyings and their correlation matrix. The processes are stored in the corresponding data member, whereas the correlation is not: instead, the process precomputes and stores its square root.<sup>6</sup> The constructor also registers with each process, in order to forward any notifications they might send.

Most other methods, such as `initialValues`, `drift`, or `expectation`, loop over the stored processes calling their corresponding methods and collecting the results in an array. The `diffusion` method also loops over the processes, but combines the results with the correlation: it multiplies each row of its square root by the diffusion term of the corresponding process, and returns the results (if you multiply it by its transposed, you'll find the familiar terms  $\sigma_i \rho_{ij} \sigma_j$  of the covariance matrix). The `stdDeviation` method does the same, but using the standard deviation of the underlying processes which also include the passed  $\Delta t$ .

This leaves us with the `evolve` method. If we knew that all the processes behaved reasonably (i.e., by adding the calculated variations to the values of their variables) we might just inherit the default implementation which takes the results of the `expectation` and `stdDeviation` methods, multiplies the latter by the array of random variates, and adds the two terms. However, we don't have such guarantee, and the method needs to be implemented in a different way. First, it deals with the correlation by multiplying the Gaussian variates by its square root, thus obtaining an array of correlated random variates. Then, it calls the `evolve` method of each one-dimensional process with the respective arguments and collects the results.

### 6.1.3 Random path generators

Before tackling path generation proper, we need a last basic component: a structure to hold the path itself. The listing that follows shows the `Path` class, which models a random path for a single variable. It contains the `TimeGrid` instance and the `Array` holding the node times and values, respectively, and provides methods to access and iterate over them. The method names are usually chosen to follow

<sup>6</sup>That would be its matricial square root; that is,  $\sqrt{A} = B$  if  $BB^T = A$ .

those of the standard containers, allowing for a more familiar interface; methods with domain-based names such as `time` are also available.

Interface of the `Path` and `MultiPath` classes.

---

```

class Path {
public:
    Path(const TimeGrid& timeGrid, const Array& values);
    bool empty() const;
    Size length() const;
    Real operator[](Size i) const;
    Real at(Size i) const;
    Real& operator[](Size i);
    Real& at(Size i);
    Real front() const;
    Real& front();
    Real back() const;
    Real& back();
    Real value(Size i) const;
    Real& value(Size i);
    Time time(Size i) const;
    const TimeGrid& timeGrid() const;
    typedef Array::const_iterator iterator;
    typedef Array::const_reverse_iterator reverse_iterator;
    iterator begin() const;
    iterator end() const;
    reverse_iterator rbegin() const;
    reverse_iterator rend() const;
private:
    TimeGrid timeGrid_;
    Array values_;
};

class MultiPath {
public:
    MultiPath(const std::vector<Path>& multiPath);
    // ...more constructors...
    Size assetNumber() const;
    Size pathSize() const;
    const Path& operator[](Size j) const;
    const Path& at(Size j) const;
    Path& operator[](Size j);
    Path& at(Size j);
private:

```

---

```
    std::vector<Path> multiPath_;
};
```

---

The `MultiPath` class, also shown in the same listing, holds paths for a number of underlying variables. It is basically a glorified container, holding a vector of `Path` instances and providing a few additional inspectors to uphold the law of Demeter.<sup>7</sup> The simplicity of the implementation is at the expense of some space: by storing a vector of `Path` instances, we're storing  $N$  identical copies of the time grid.

### Aside: access patterns.

The basic inspectors in the `Path` class, such as `operator[]` and the iterator interface, return the values  $S_i$  of the underlying variable rather than the pairs  $(t_i, S_i)$  including the node times. While it could well be expected that `p[i]` return the whole node, we thought that in the most common cases a user would want just the value, so we optimized such access (it's some kind of Huffman-coding principle, in which the most common operations should require the least typing; the Perl language shines at this, or so I'm told).

In a sad note about reuse (or lack thereof) if we were to add to the `Path` class a few methods to return whole nodes, we probably wouldn't use `std::pair<Time, Real>` to hold the data, but an inner `Node` struct. This is not because we instinctively leap at the most complex implementation—even though it may seem so at times—but because code such as `p.node(i).second` is not self-describing (wait, is `second` the time or the value?) The advantage of writing the above as `p.node(i).value` would offset the cost of defining a `Node` struct.

What remains to be done for path generation is now to connect the dots between the classes that I described so far. The logic for doing so is implemented in the `MultiPathGenerator` class template, sketched in the next listing; I'll leave it to you to decide whether this makes it an instance of the Factory pattern.

Sketch of the `MultiPathGenerator` class template.

---

```
template <class GSG>
class MultiPathGenerator {
public:
    typedef Sample<MultiPath> sample_type;
    MultiPathGenerator(const shared_ptr<StochasticProcess>&,
                       const TimeGrid&,
                       GSG generator,
                       bool brownianBridge = false);
    const sample_type& next() const { return next(false); }
```

---

<sup>7</sup>For instance, if `p` is a `MultiPath` instance, the additional methods allow us to write `p.pathSize()` instead of `p[0].length()`. Besides being uglier, the latter might also suggest that `p[1].length()` could be different.

```

    const sample_type& antithetic() const { return next(true); }

private:
    const sample_type& next(bool antithetic) const;
    bool brownianBridge_;
    shared_ptr<StochasticProcess> process_;
    GSG generator_;
    mutable sample_type next_;

};

template <class GSG>
const typename MultiPathGenerator<GSG>::sample_type&
MultiPathGenerator<GSG>::next(bool antithetic) const {
    if (brownianBridge_) {
        QL_FAIL("Brownian bridge not supported");
    } else {
        typedef typename GSG::sample_type sequence_type;
        const sequence_type& sequence_ =
            antithetic ? generator_.lastSequence()
                        : generator_.nextSequence();
        Size m = process_->size(), n = process_->factors();
        MultiPath& path = next_.value;
        Array asset = process_->initialValues();
        for (Size j=0; j<m; j++)
            path[j].front() = asset[j];
        Array temp(n);
        next_.weight = sequence_.weight;

        TimeGrid timeGrid = path[0].timeGrid();
        Time t, dt;
        for (Size i = 1; i < path.pathSize(); i++) {
            Size offset = (i-1)*n;
            t = timeGrid[i-1];
            dt = timeGrid.dt(i-1);
            if (antithetic)
                std::transform(sequence_.value.begin()+offset,
                              sequence_.value.begin()+offset+n,
                              temp.begin(),
                              std::negate<Real>());
            else
                std::copy(sequence_.value.begin()+offset,
                          sequence_.value.begin()+offset+n,
                          temp.begin());
    }
}

```

---

```

        asset = process_->evolve(t, asset, dt, temp);
        for (Size j=0; j<m; j++)
            path[j][i] = asset[j];
    }
    return next_;
}

```

---

Its constructor takes and stores the stochastic process followed by the variable to be modeled; the time grid to be used for generating the path nodes; a generator of random Gaussian sequences, which must be of the right dimension for the job (that is, at each draw it must return  $N \times M$  numbers for  $N$  factors and  $M$  steps); and a boolean flag to specify whether or not it should use Brownian bridging, which at this time cannot be set to `true` and with which we're unfortunately stuck for backward compatibility.

Oh, a brief note before continuing. I won't describe here a few techniques that can be used to improve the accuracy of Monte Carlo simulations, and which are implemented in our library: these include the Brownian bridge I just mentioned and a couple of variance-reduction techniques that will come into play later, that is, antithetic variates and control variates. Again, I refer you to Glasserman or Jäckel (or, well, the Internet) if you're not already familiar with them. I'll describe the code assuming that you have some idea of how they work.

Back to the `MultiPathGenerator` class. Its interface provides a `next` method, returning a new random path (well, multipath, but cut me some slack here), and an `antithetic` method, returning the antithetic path of the one last returned by `next`,<sup>8</sup> that is, the path generated by replacing the used random numbers with their opposites. Both methods forward to a private method `next(bool)`, which implements the actual logic. If the `brownianBridge` flag was set to `true`, the method bails out by raising an exception since the functionality is not yet implemented. Following the general principles of C++, which suggest to fail as early as possible, we should probably move the check to the constructor (or better yet, implement the thing; send me a patch if you do). As it is now, one can build a path generator and have the constructor succeed, only to find out later that the instance is unusable.

If Brownian bridging is not required, the method gets to work. First, it retrieves the random sequence it needs: a new one if we asked for a new path, or the latest sequence we used if we asked for an antithetic path. Second, it performs some setup. It gets a reference to the path to be built (which is a data member, not a temporary; more on this later); retrieves the array of the initial values of the variables and copies them at the beginning of the path; creates an array to hold the subsets of the random variables that will be used at each step; and retrieves the time grid to be used. Finally, it puts the pieces together. At each step, it takes as starting point the values at the previous one (it uses the same array where it stored the initial values, so the first step is covered, too); it retrieves from the time grid the current time `t` and the interval `dt` over which to evolve the variables; it copies

---

<sup>8</sup>Unfortunately, the correctness of the `antithetic` method depends on its being called at the correct time, that is, after a call to `next`; but I can't think of any acceptable way out of this.

the subset of the random variables that it needs (the first  $N$  for the first step, the second  $N$  for the second step, and so on,  $N$  being the number of factors; also, it negates them if it must generate an antithetic path); and to close the circle, it calls the `evolve` method of the process and copies the returned values at the correct places in the paths.

As I mentioned, the returned path is stored as a data member of the generator instance. This saves quite a few allocations and copies; but of course it also makes it impossible to share a path generator between threads, since they would race for the same storage. However, that's just the last nail in the coffin. Multithreading was pretty much out of the picture as soon as we stored the Gaussian number generator as a data member; the path generator can only be as thread-safe as its RNG—that is, not much, since most (if not all) of them store state. A much easier strategy to distribute work among threads would be to use different `MultiPathGenerator` instances, each with its own RNG. The problem remains of creating RNGs returning non-overlapping sequences; for algorithms that allow it (such as Sobol's) one can tell each instance to skip ahead a given number of draws.

### Aside: stepping on one's own toes.

Unfortunately, there's quite a bit of memory allocation going on during multi-path generation. The obvious instances (the two `Arrays` being created to store the current variable values and the current subset of the random numbers) could be avoided by storing them as data members; but on the one hand, we would have yet more `mutable` members, which is something we should use sparingly; and on the other hand, the allocation of the two arrays only happens once in a whole path generation, so it might not make a lot of difference.

Instead, the worst offender might be the underlying process, which at each step creates an `Array` instance to be returned from its `evolve` method. How can we fix this? In this case, I wouldn't return a reference to a data member if I can help it: the `StochasticProcess` class might just work in a multithreaded environment (well, if market data don't change during calculation, or if at least the process is kept frozen) and I'd rather not ruin it. Another possibility might be that the `evolve` method take a reference to the output array as an argument; but then, we (and for *we*, I mean whoever will implement a stochastic process) would have to be very carefully about aliasing.

The sad fact is that the natural way to write the `evolve` method would usually be something like (in pseudocode, and in the case of, say, two variables)

```
void evolve(const Array& x, Time t, Time dt,
             const Array& w, Array& y) {
    y[0] = f(x[0], x[1], t, dt, w[0], w[1]);
    y[1] = g(x[0], x[1], t, dt, w[0], w[1]);
}
```

where `x` is the input array and `y` is the output. However, it would be just as natural for a user to write

```
p->evolve(x,t,dt,w,x);
```

(that is, to pass the same  $x$  as both input and output) meaning to replace the old values of the variables with the new ones.

Well, you see the catch. When the first instruction in `evolve` writes into  $y[0]$ , it actually writes into  $x[0]$ . The second instruction then calls `g` with a value of  $x[0]$  which is not the intended one. Hilarity ensues.

Fixing this requires either the developer to make each process safe from aliasing, by writing

```
a = f(...); b = g(...); y[0] = a; y[1] = b;
```

or something to this effect; or the user to be mindful of the risk of aliasing, and never to pass the same array as both input and output. Either solution requires more constant care than I credit myself with.

So, where does this leave us? For the time being, the current interface might be an acceptable compromise. One future possibility is that we take away the allocation cost at the origin, by using some kind of allocation pool inside the `Array` class. As usual, we're open to suggestions.

To close the section, I'll mention that the library also provides a `PathGenerator` class for 1-D stochastic processes. It has some obvious differences (it calls the `StochasticProcess1D` interface and it creates `Paths` instead of `MultiPaths`; also, it implements Brownian bridging, which is a lot easier in the 1-D case) but other than that, it has the same logic as `MultiPathGenerator` and doesn't warrant being described in detail here.

## 6.2 Pricing on a path

There's not much to say on the `PathPricer` class template, shown in the listing below. It defines the interface that must be implemented by a path pricer in order to be used in a Monte Carlo model: an `operator()` taking a path and returning some kind of result. It is basically a function interface: in fact, if we were to write the framework now instead of almost twenty years ago, I'd just use `std::function` and dispense with `PathPricer` altogether.

Interface of the `PathPricer` class template.

---

```
template<class PathType, class ValueType=Real>
class PathPricer : public unary_function<PathType,ValueType> {
public:
    virtual ~PathPricer() {}
    virtual ValueType operator()(const PathType& path) const=0;
};
```

---

The template arguments generalize on both the type of the argument, which can be a `Path`, a

`MultiPath`, or a user-defined type if needed;<sup>9</sup> and the type of the result, which defaults to a simple `Real` (usually the value of the instrument) but might be an array of results, or some kind of structure. On the one hand, one might like to have the choice: an `Array` would be more appropriate for returning a set of homogeneous results, such as the amounts of a series of coupons, while a structure with named fields would be better suited for inhomogeneous results such as the value and the several Greeks of an instrument. On the other hand, the results will need to be combined, and therefore they'll need some basic algebra; the `Array` class already provides it, which makes it ready to use, whereas a results structure would need to define some operators in order to play well with the facilities described in the next section.

## 6.3 Putting it all together

Having read all about the different pieces of a model, you'd expect me to start assembling them and finally get a working Monte Carlo model. However, before doing so, we still have the task of picking a specific set of pieces among those available in our toolbox. This will be the subject of the next subsection—after which we'll build the model, developer's honor.

### 6.3.1 Monte Carlo traits

As I mentioned before, there's a number of choices we have to make in order to implement a Monte Carlo model. Depending on the dimension of the problem and the stochastic process we choose, we can use one- or multi-dimensional paths and generators; or we can use pseudo-random numbers or low-discrepancy sequences (and even though I haven't listed them, there's quite a few algorithms available for either type).

For at least one of those choices, dynamic polymorphism is not an option: one-dimensional and multi-dimensional generators can't have a common interface, since their methods have different input and return types. Therefore, we'll go with static polymorphism and supply the required classes to the model as template arguments.

The problem is, this can result very quickly in unwieldy declarations; the sight of something like

```
MonteCarloModel<
    MultiPathGenerator<
        InverseCumulativeRsg<SobolRsg, InverseCumulativeNormal> > >
```

can bring shudders to the most seasoned developer. Sure, `typedefs` can help; by using them, a user might assign mnemonics to default choices and shorten the declaration. Default template arguments might alleviate the pain, too. However, we went for a mechanism that also allowed us on the one hand, to define mnemonics for commonly used groups of related choices; and on the other hand, to add helper methods to the mix. To do this, we decided to use traits.

---

<sup>9</sup>For instance, when pricing instruments with an early-termination feature, one might want to save time by using a path class that generates nodes lazily and only if they are actually used.

The following listing shows the default traits classes for pseudo-random and low-discrepancy number generation.

Example of random-number generation traits.

---

```

template <class URNG, class IC>
struct GenericPseudoRandom {
    typedef URNG urng_type;
    typedef InverseCumulativeRng<urng_type,IC> rng_type;
    typedef RandomSequenceGenerator<urng_type> ursg_type;
    typedef InverseCumulativeRsg<ursg_type,IC> rsg_type;
    enum { allowsErrorEstimate = 1 };
    static rsg_type make_sequence_generator(Size dimension,
                                              BigNatural seed) {
        ursg_type g(dimension, seed);
        return rsg_type(g);
    }
};

typedef GenericPseudoRandom<
    MersenneTwisterUniformRng,
    InverseCumulativeNormal>
PseudoRandom;

template <class URSG, class IC>
struct GenericLowDiscrepancy {
    typedef URSG ursg_type;
    typedef InverseCumulativeRsg<ursg_type,IC> rsg_type;
    enum { allowsErrorEstimate = 0 };
    static rsg_type make_sequence_generator(Size dimension,
                                              BigNatural seed) {
        ursg_type g(dimension, seed);
        return rsg_type(g);
    }
};

typedef GenericLowDiscrepancy<
    SobolRsg,
    InverseCumulativeNormal>
LowDiscrepancy;

```

---

First comes the `GenericPseudoRandom` class template. It takes as template arguments the type of a uniform pseudo-random number generator and that of an inverse-cumulative function object, and

builds a number of other types upon it. The type of the passed generator itself is defined as `urng_type`—emphasis on “u” for uniform and “n” for number. Based on this type, it defines `rng_type`, no longer uniform since it uses the inverse-cumulative function to return numbers according to its distribution; `ursg_type`, where the “n” is replaced by “s” for sequence; and finally `rsg_type`, which generates sequences of numbers according to the passed distribution. The compile-time constant `allowErrorEstimate`, written as an enumeration to satisfy older compilers (it should really be a `static const bool`) tells us that this generator allows us to estimate the Monte Carlo error as function of the number of samples; and the helper function `make_sequence_generator` makes it easier to create a generator based on the passed inputs.

Then, we instantiate the class template with our weapons of choice. For the basic generator, that would be the `MersenneTwisterUniformRng` class; for the function object, the `InverseCumulativeNormal` class, since we’ll most often want normally distributed numbers. The resulting traits class will be our default for pseudo-random generation; fantasy being not our strong suit, we defined it as the `PseudoRandom` class.

The `GenericLowDiscrepancy` class template is defined is a similar way, but with two differences. Since low-discrepancy numbers are generated in sequences, the types for single-number generation are missing; and the enumeration tells us that we can’t forecast the statistical error we’ll get. We define the `LowDiscrepancy` traits class as the one obtained by selecting the `SobolRsg` class as our generator and, again, the `InverseCumulativeNormal` class as our function object.

Finally, we defined a couple of traits classes to hold types related to specific Monte Carlo functionality, such as the types of used paths, path generators, and path pricers. They are shown in the next listing: the `SingleVariate` class holds the types we need for 1-D models, while the `MultiVariate` class holds the types for multi-dimensional ones. They are both template classes, and take as their template argument a traits class for random-number generation.

Example of Monte Carlo traits.

---

```
template <class RNG = PseudoRandom>
struct SingleVariate {
    typedef RNG rng_traits;
    typedef Path path_type;
    typedef PathPricer<path_type> path_pricer_type;
    typedef typename RNG::rsg_type rsg_type;
    typedef PathGenerator<rsg_type> path_generator_type;
    enum { allowsErrorEstimate = RNG::allowsErrorEstimate };
};

template <class RNG = PseudoRandom>
struct MultiVariate {
    typedef RNG rng_traits;
    typedef MultiPath path_type;
    typedef PathPricer<path_type> path_pricer_type;
    typedef typename RNG::rsg_type rsg_type;
```

---

```

typedef MultiPathGenerator<rsg_type> path_generator_type;
enum { allowsErrorEstimate = RNG::allowsErrorEstimate };

};
```

---

By combining the provided RNG and Monte Carlo traits (or any traits classes that one might want to define, if one wants to use any particular type) not only we can provide a model with all the necessary information, but we can do it with a simpler and more mnemonic syntax, such as

```
MonteCarloModel<SingleVariate, LowDiscrepancy>;
```

the idea being to move some complexity from users to developers. We have to use some template tricks to get the above to work, but when it does, it's a bit more readable (and writable) for users. But that's for next section, in which we finally assemble a Monte Carlo model.

### 6.3.2 The Monte Carlo model

The listing below shows the `MonteCarloModel` class, which is the low-level workhorse of Monte Carlo simulations.

Implementation of the `MonteCarloModel` class template.

---

```

template <template <class> class MC, class RNG,
                      class S = Statistics>

class MonteCarloModel {
    public:
        typedef MC<RNG> mc_traits;
        typedef RNG rng_traits;
        typedef typename MC<RNG>::path_generator_type
                           path_generator_type;
        typedef typename MC<RNG>::path_pricer_type path_pricer_type;
        typedef typename path_generator_type::sample_type
                           sample_type;
        typedef typename path_pricer_type::result_type result_type;
        typedef S stats_type;

    MonteCarloModel(
        const shared_ptr<path_generator_type>& pathGenerator,
        const shared_ptr<path_pricer_type>& pathPricer,
        const stats_type& sampleAccumulator,
        bool antitheticVariate,
        const shared_ptr<path_pricer_type>& cvPathPricer
            = shared_ptr<path_pricer_type>(),
        result_type cvOptionValue = result_type(),
```

```

const shared_ptr<path_generator_type>& cvGenerator
    = shared_ptr<path_generator_type>());

void addSamples(Size samples);
const stats_type& sampleAccumulator(void) const;
private:
    shared_ptr<path_generator_type> pathGenerator_;
    shared_ptr<path_pricer_type> pathPricer_;
    stats_type sampleAccumulator_;
    bool isAntitheticVariate_;
    shared_ptr<path_pricer_type> cvPathPricer_;
    result_type cvOptionValue_;
    bool isControlVariate_;
    shared_ptr<path_generator_type> cvPathGenerator_;
};

template <template <class> class MC, class RNG, class S>
MonteCarloModel<MC,RNG,S>::MonteCarloModel(
    const shared_ptr<path_generator_type>& pathGenerator,
    ...other arguments...)
: pathGenerator_(pathGenerator), ...other data... {
    if (!cvPathPricer_)
        isControlVariate_ = false;
    else
        isControlVariate_ = true;
}

template <template <class> class MC, class RNG, class S>
void MonteCarloModel<MC,RNG,S>::addSamples(Size samples) {
    for (Size j = 1; j <= samples; j++) {
        sample_type path = pathGenerator_->next();
        result_type price = (*pathPricer_)(path.value);

        if (isControlVariate_) {
            if (!cvPathGenerator_) {
                price += cvOptionValue_-(*cvPathPricer_)(path.value);
            } else {
                sample_type cvPath = cvPathGenerator_->next();
                price +=
                    cvOptionValue_-(*cvPathPricer_)(cvPath.value);
            }
        }
    }
}

```

```

if (isAntitheticVariate_) {
    path = pathGenerator_->antithetic();
    result_type price2 = (*pathPricer_)(path.value);
    if (isControlVariate_) {
        ... adjust the second price as above
    }
    sampleAccumulator_.add((price+price2)/2.0, path.weight);
} else {
    sampleAccumulator_.add(price, path.weight);
}
}
}

```

---

It brings together path generation, pricing and statistics, and as such takes template arguments defining the types involved: a `MC` traits class defining types related to the simulation, a `RNG` traits class describing random-number generation, and a statistics class `S` defaulting to the `Statistics` class.<sup>10</sup> The `MC` class is a template template argument, so that it can be fed the `RNG` traits (as shown in the previous section; see for instance the `MultiVariate` class).

The class defines aliases for a few frequently used types; most of them are extracted from the traits, by instantiating the `MC` class template with the `RNG` class. The resulting class provides the types of the path generator and the path pricer to be used; from those, in turn, the type of the sample paths and that of the returned prices can be obtained.

The constructor takes the pieces that will be made to work together; at least a path generator (well, a pointer to one, but you'll forgive me for not spelling out all of them), a path pricer, and an instance of the statistics class, as well as a boolean flag specifying whether to use antithetic variates. Then, there are a few optional arguments related to control variates: another path pricer, the analytic value of the control variate, and possibly another path generator. Optional arguments might not be the best choice, since they make it possible to pass a control variate path pricer and not the corresponding analytic value; it would have been safer to have a constructor with no control variate arguments, and another constructor with both path pricer and analytic value being mandatory and with an optional path generator. However, the current version saves a few lines of code. The constructor copies the passed arguments into the corresponding data members, and sets another boolean flag based on the presence or the lack of the control-variate arguments.

The main logic is implemented in the `addSamples` method. It's basically a loop that draws a path, prices, and adds the result to the statistics; but it includes a bit of complication in order to take care of variance reduction. It takes the number of samples to add; for each of them, it asks the path generator for a path, passes the path to the pricer, and gets back a price. In the simplest case, that's all there is to it; the price can be added to the statistics (together with the corresponding weight, also returned from the generator) and the loop can start the next iteration. If the user passed control-variate data,

<sup>10</sup>Don't worry. I'm not going off another tangent, even though an early outline of this chapter had a section devoted to statistics. If you're interested, the `Statistics` class is in appendix A.

instead, things get more interesting. If no path generator were specified, we pass to the alternate pricer the same path we used for the main one; otherwise, we ask the second generator for a path and we use that one. In both cases, we adjust the baseline price by subtracting the simulated price of the control and adding its analytic value.

It's not over yet. If the user also asked for antithetic variates, we repeat the same dance (this time asking the generator, or the generators, for the paths antithetic to the ones we just used) and we add to the statistics the average of the regular and antithetic prices; if not, we just add the price we obtained on the original paths. Lather, rinse, and repeat until the required number of samples is reached.

Finally, the full results (mean price and whatnot) can be obtained by calling the `sampleAccumulator` method, which returns a reference to the stored statistics. “Accumulator” is STL lingo; we should probably have used a method name taken from the financial domain instead. Such as, I don't know, “statistics.” Oh well.

### 6.3.3 Monte Carlo simulations

Up one step in complexity and we get to the `McSimulation` class template, sketched in the next listing. Its job is to drive the simple-minded `MonteCarloModel` towards a goal, be it a required accuracy or number of samples. It can be used (and it was designed) as a starting point to build a pricing engine.

Sketch of the `McSimulation` class template.

---

```
template <template <class> class MC, class RNG,
          class S = Statistics>
class McSimulation {
public:
    typedef
        typename MonteCarloModel<MC,RNG,S>::path_generator_type
        path_generator_type;
    typedef typename MonteCarloModel<MC,RNG,S>::path_pricer_type
        path_pricer_type;
    typedef typename MonteCarloModel<MC,RNG,S>::stats_type
        stats_type;
    typedef typename MonteCarloModel<MC,RNG,S>::result_type
        result_type;
    virtual ~McSimulation() {}
    result_type value(Real tolerance,
                      Size maxSamples = QL_MAX_INTEGER,
                      Size minSamples = 1023) const;
    result_type valueWithSamples(Size samples) const;
    void calculate(Real requiredTolerance,
                   Size requiredSamples,
```

```

                Size maxSamples) const;
const stats_type& sampleAccumulator(void) const;
protected:
    McSimulation(bool antitheticVariate,
                 bool controlVariate);
    virtual shared_ptr<path_pricer_type> pathPricer() const = 0;
    virtual shared_ptr<path_generator_type> pathGenerator()
                                         const = 0;
    virtual TimeGrid timeGrid() const = 0;
    virtual shared_ptr<path_pricer_type>
controlPathPricer() const;
    virtual shared_ptr<path_generator_type>
controlPathGenerator() const;
    virtual result_type controlVariateValue() const;
    virtual shared_ptr<PricingEngine> controlPricingEngine()
                                         const;
mutable shared_ptr<MonteCarloModel<MC,RNG,S> > mcModel_;
bool antitheticVariate_, controlVariate_;
};

template <template <class> class MC, class RNG, class S>
typename McSimulation<MC,RNG,S>::result_type
McSimulation<MC,RNG,S>::value(Real tolerance,
                                Size maxSamples,
                                Size minSamples) const {
    ...
    Real error = mcModel_->sampleAccumulator().errorEstimate();
    while (error > tolerance) {
        QL_REQUIRE(sampleNumber < maxSamples, ...);
        Real order = (error*error)/(tolerance*tolerance);
        Size nextBatch =
            std::max<Size>(sampleNumber*order*0.8-sampleNumber,
                            minSamples));
        nextBatch = std::min(nextBatch, maxSamples-sampleNumber);
        sampleNumber += nextBatch;
        mcModel_->addSamples(nextBatch);
        error = mcModel_->sampleAccumulator().errorEstimate();
    }
    return mcModel_->sampleAccumulator().mean();
}

template <template <class> class MC, class RNG, class S>
void McSimulation<MC,RNG,S>::calculate(Real requiredTolerance,

```

```

        Size requiredSamples,
        Size maxSamples) const {
    if (!controlVariate_) {
        mcModel_ = shared_ptr<MonteCarloModel<MC,RNG,S>>(
            new MonteCarloModel<MC,RNG,S>(
                pathGenerator(), pathPricer(),
                S(), antitheticVariate_));
    } else {
        ... // same as above, but passing the control-variate args, too.
    }

    if (requiredTolerance != Null<Real>())
        this->value(requiredTolerance, maxSamples);
    else
        this->valueWithSamples(requiredSamples);
}

```

---

`McSimulation` follows the Template Method pattern. It asks the user to implement a few pieces of behavior, and in return it provides generic logic to instantiate a Monte Carlo model and run a simulation. Derived classes must define at least the `pathPricer` method, that returns the path pricer to be used; the `pathGenerator` method, that returns the path generator; and the `timeGrid` method, that returns the grid describing the nodes of the simulation. Other methods, returning the objects to be used for control variates, might or might not be defined; `McSimulation` provides default implementations that return null values, so derived classes that don't want to use the control variate technique can just forget about it.

Based on such methods, `McSimulation` provides most of the behavior needed by an engine. Its constructor takes two boolean flags specifying whether it should use either antithetic or control variates; the second will only matter if the derived class implements the required methods.

The `value` method adds samples to the underlying model until the estimated error matches a required tolerance.<sup>11</sup> It looks at the current number of samples  $n$  and the current error  $\epsilon$ , estimates the number of samples  $N$  that will be needed to reach the given tolerance  $\tau$  as  $N = n \times \epsilon^2 / \tau^2$  (since of course  $\epsilon \propto 1/\sqrt{n}$ ), and adds a new batch of  $N - n$  samples that gets the total closer to the estimated number; then it assesses the error again, and repeats the process as needed.

The `valueWithSamples` method just adds a batch of samples so that their total number matches the required one; its implementation is not shown here because of space constraints, but is written as you might expect.

Finally, the `calculate` method runs a complete simulation. It takes as arguments either a required tolerance or a required number of samples,<sup>12</sup> as well as a maximum number of samples; it instantiates a `MonteCarloModel` instance, with the `controlVariate_` flag determining whether to pass the

<sup>11</sup>Of course, this needs an error estimate, so it won't work with low-discrepancy methods.

<sup>12</sup>One of the arguments must be null, but not both.

control-variate arguments to the constructor; and it calls either the `value` or the `valueWithSamples` method, depending on what goal was required.

In next section, I'll show an example of how to use the `McSimulation` class to build a pricing engine; but before that, let me point out a few ways in which it could be improved.

First of all, it currently implements logic to run a simulation based on two criteria (accuracy or total number of samples) but of course, more criteria could be added. For instance, one could run a simulation until a given clock time is elapsed; or again, one could add several batches of samples, look at the result after each one, and stop when convergence seems to be achieved. This suggests that the hard-coded `value` and `valueWithSamples` methods could be replaced by an instance of the Strategy pattern, and the switch between them in the `calculate` method by just a call to the stored strategy.

In turn, this would also remove the current ugliness in `calculate`: instead of passing the whole set of arguments for both calculations and giving most of them to a null value (like in the good old days, where languages could not overload methods) one would pass only the required arguments to the strategy object, and then the strategy object to `calculate`.

Finally, the presence of `controlPricingEngine` method is a bit of a smell. The implementation of `McSimulation` doesn't use it, so it's not strictly a part of the Template Method pattern and probably shouldn't be here. However, a couple of other classes (both inheriting from `McSimulation`, but otherwise unrelated) declare it and use it to implement the `controlVariateValue` method; therefore, leaving it here might not be the purest of designs but prevents some duplication.

## Aside: synchronized walking.

Both the `MonteCarloModel` and `McSimulation` class templates allow one to define an alternate path generator for control variates. Note, however, that this feature should be used with some caution. For this variance-reduction technique to work, the control-variate paths returned from the second generator must be correlated with the regular ones, which basically means that the two path generators must use two identical random-number generators: same kind, dimensionality, and seed (if any).

Unfortunately, this constraint rules out quite a few possibilities. For instance, if you're using a stochastic volatility model, such as the Heston model, you might be tempted to use the Black-Scholes model as control variate. No such luck: for  $n$  time steps, the Heston process requires  $2n$  random numbers ( $n$  for the stock price and  $n$  for its volatility) while the Black-Scholes process just needs  $n$ . This makes it impossible to keep the two corresponding path generators in sync.

In practice, you'll have a use case for the alternate path generator if you have a process with a number of parameters which is not analytically tractable in the generic case, but has a closed-form solution for your option value if some of the parameters are null. If you're so lucky, you can use a fully calibrated process to instantiate the main path generator, and another instance of the process with the null parameters to generate control-variate paths.

### 6.3.4 Example: basket option

To close this chapter, I'll show and discuss an example of how to build a pricing engine with the Monte Carlo machinery I described so far. The instrument I'll use is a simple European option on a basket of stocks, giving its owner the right to buy or sell the whole basket at a given price; the quantities of each of the stocks in the basket are also specified by the contract.

For brevity, I won't show the implementation of the instrument class itself, `BasketOption`.<sup>13</sup> It would be quite similar to the `VanillaOption` class I've shown in [chapter 2](#), with the addition of a data member for the quantities (added to both the instrument and its arguments class). Also, I won't deal with the Greeks; but if the `BasketOption` class were to define them, methods such as `delta` and `gamma` would return a vector.

The main class in this example is the one implementing the pricing engine. It is the `MCEuropeanBasketEngine` class template, shown in the listing below. As expected, it inherits publicly from the `BasketOption::engine` class; however, it also inherits privately from the `McSimulation` class template.

Implementation of the `MCEuropeanBasketEngine` class template.

---

```
template <class RNG = PseudoRandom, class S = Statistics>
class MCEuropeanBasketEngine
    : public BasketOption::engine,
    private McSimulation<MultiVariate, RNG, S> {
public:
    typedef McSimulation<MultiVariate, RNG, S> simulation_type;
    typedef typename simulation_type::path_generator_type
        path_generator_type;
    ... // same for the other defined types
    MCEuropeanBasketEngine(
        const shared_ptr<StochasticProcess>&,
        const Handle<YieldTermStructure>& discountCurve,
        Size timeSteps,
        Size timeStepsPerYear,
        bool antitheticVariate,
        Size requiredSamples,
        Real requiredTolerance,
        Size maxSamples,
        BigNatural seed);
    void calculate() const {
        simulation_type::calculate(requiredTolerance_,
            requiredSamples_,
            maxSamples_);
        const S& stats = this->mcModel_->sampleAccumulator();
        results_.value = stats.mean();
```

---

<sup>13</sup>This class is not the same as the `BasketOption` class implemented in QuantLib. The one used here is simplified for illustration purposes.

```

    if (RNG::allowsErrorEstimate)
        results_.errorEstimate = stats.errorEstimate();
}
private:
    TimeGrid timeGrid() const;
    shared_ptr<path_generator_type> pathGenerator() const;
    shared_ptr<path_pricer_type> pathPricer() const;
    shared_ptr<StochasticProcess> process_;
    Handle<YieldTermStructure> discountCurve_;
    Size timeSteps_, timeStepsPerYear_;
    Size requiredSamples_;
    Size maxSamples_;
    Real requiredTolerance_;
    BigNatural seed_;
};

template <class RNG, class S>
TimeGrid MCEuropeanBasketEngine<RNG,S>::timeGrid() const {
    Time T = process_->time(arguments_.exercise->lastDate());
    if (timeSteps_ != Null<Size>()) {
        return TimeGrid(T, timeSteps_);
    } else if (timeStepsPerYear_ != Null<Size>()) {
        Size steps = timeStepsPerYear_*T;
        return TimeGrid(T, std::max<Size>(steps, 1));
    } else {
        QL_FAIL("time steps not specified");
    }
}

template <class RNG, class S>
shared_ptr<typename MCEuropeanBasketEngine<RNG,S>::
    path_generator_type>
MCEuropeanBasketEngine<RNG,S>::pathGenerator() const {
    Size factors = process_->factors();
    TimeGrid grid = timeGrid();
    Size steps = grid.size() - 1;
    typename RNG::rsg_type gen =
        RNG::make_sequence_generator(factors*steps,
                                      seed_);
    return shared_ptr<path_generator_type>(
        new path_generator_type(process_, grid, gen));
}

```

```

template <class RNG, class S>
shared_ptr<typename MCEuropeanBasketEngine<RNG,S>::
    path_pricer_type>
MCEuropeanBasketEngine<RNG,S>::pathPricer() const {
    Date maturity = arguments_.exercise->lastDate();
    return shared_ptr<path_pricer_type>(
        new EuropeanBasketPathPricer(
            arguments_.payoff, arguments_.quantities,
            discountCurve_->discount(maturity)));
}

```

---

In idiomatic C++, the use of private inheritance denotes an “is implemented in terms of” relationship. We don’t want public inheritance here, since that would imply an “is a” relationship; in our conceptual model, `MCEuropeanBasketEngine` is a pricing engine for the basket option and not a simulation that could be used on its own. The use of multiple inheritance is shunned by some, and in fact it could be avoided in this case; our engine might use composition instead, and contain an instance of `McSimulation`. However, that would require inheriting a new simulation class from `McSimulation` in order to implement its purely virtual methods (such as `pathGenerator` or `pathPricer`) and would have the effect to make the design of the engine more complex; whereas, by inheriting from `McSimulation`, we can define such methods in the engine itself.<sup>14</sup> Finally, note the template parameters: we leave to the user the choice of the RNG traits, but since we’re modeling a basket option we specify the `MultiVariate` class as the Monte Carlo traits.

The constructor (whose implementation is not shown for brevity) takes the stochastic process for the underlyings, a discount curve, and a number of parameters related to the simulation. It copies the arguments into the corresponding data members (except for the `antitheticVariate` flag, which is passed to the `McSimulation` constructor) and registers the newly-built instance as an observer of both the stochastic process and the discount curve.

The `calculate` method, required by the `PricingEngine` interface, calls the method by the same name from `McSimulation` and passes it the needed parameters; then, it retrieves the statistics and stores the mean value and, if possible, the error estimate. This behavior is not specific of this particular option, and in fact it could be abstracted out in some generic `McEngine` class; but this would muddle the conceptual model. Such a class would inherit from `McSimulation`, but it could be reasonably expected to inherit from `PricingEngine`, too. However, if we did that, our engine would inherit from both `McEngine` and `BasketOption::engine`, leading to the dreaded inheritance diamond. On the other hand, if we didn’t inherit it from `PricingEngine` (calling it `McEngineAdapter` or something) it would add more complexity to the inheritance hierarchy, since it would be an additional layer between `McSimulation` and our engine, and wouldn’t remove all the scaffolding anyway: our engine would still need to define a `calculate` method forwarding to the one in the adapter. Thus, I guess we’ll leave it at that.

<sup>14</sup>In order to avoid this dilemma, we would have to rewrite the `McSimulation` class so that it doesn’t use the Template Method pattern; that is, we should make it a concrete class which would be passed the used path generator and pricer as constructor arguments. Apart from breaking backward compatibility, I’m not sure this would be worth the hassle.

The rest of the listing shows the three methods required to implement the `McSimulation` interface and turn our class into a working engine. The `timeGrid` method builds the grid used by the simulation by specifying the end time (given by the exercise date) and the number of steps. Depending on the parameters passed to the engine constructor, they might have been specified as a given total number (the first `if` clause) or a number per year (the second). In either case, the specification is turned into a total number of steps and passed to the `TimeGrid` constructor. If neither was specified (the `else` clause) an error is raised.

The `pathGenerator` method asks the process for the number of factors, determines the number of time steps from the result of the `timeGrid` method I just described, builds a random-sequence generator with the correct dimensionality (which of course is the product of the two numbers above) and uses it together with the underlying process and the time grid to instantiate a multi-path generator.

Finally, the `pathPricer` method collects the data required to determine the payoff on each path—that is, the payoff object itself, the quantities and the discount at the exercise date, that the method precalculates—and builds an instance of the `EuropeanBasketPathPricer` class, sketched in the next listing.

Sketch of the `EuropeanMultiPathPricer` class.

---

```
class EuropeanBasketPathPricer : public PathPricer<MultiPath> {
public:
    EuropeanBasketPathPricer(const shared_ptr<Payoff>& payoff,
                           const vector<Real>& quantities,
                           DiscountFactor discount);
    Real operator()(const MultiPath& path) const {
        Real basketValue = 0.0;
        for (Size i=0; i<quantities_.size(); ++i)
            basketValue += quantities_[i] * path[i].back();
        return (*payoff)(basketValue) * discount_;
    }
private:
    shared_ptr<Payoff> payoff_;
    vector<Real> quantities_;
    DiscountFactor discount_;
};
```

---

The path pricer stores the passed data and uses them to calculate the realized value of the option in its `operator()` overloading, whose implementation calculates the value of the basket at maturity and returns the corresponding payoff discounted to the present time. The calculation of the basket value is a straightforward loop that combines the stored quantities with the asset values at maturity, retrieved from the end points of the respective paths.

At this point, the engine is completed; but it's still a bit unwieldy to instantiate. We'd want to add a factory class with a fluent interface, so that one can write

```
engine = MakeMcEuropeanBasketEngine<PseudoRandom>(process,
                                                       discountCurve)
    .withTimeStepsPerYear(12)
    .withAbsoluteTolerance(0.001)
    .withSeed(42)
    .withAntitheticVariate();
```

but I'm not showing its implementation here. There's plenty of such examples in the library for your perusal.

Also, you'll have noted that I kept the example as simple as possible, and for that reason I avoided a few possible generalizations. For instance, another pricer might need some other specific dates besides the maturity, and the time grid should be built accordingly; or the payoff might have been path-dependent, so that the path pricer should look at several path values besides the last. But I don't think you'll have any difficulties in writing the corresponding code.

### Aside: need-to-know basis.

As usual, I didn't show in which files one should put the several classes I described. Well, as a general principle, the more encapsulation the better; thus, to begin with, helper classes should be hidden from the public interface. For instance, the `EuropeanBasketPathPricer` class is only used by the engine and could be hidden from client code. The best way would be to define such classes inside an anonymous namespace in a `.cpp` file, but that's not always possible: in our case, the engine is a class template, so that's not an option. The convention we're using in QuantLib is that helper classes that must be declared in a header file are placed in a nested namespace `detail`, and the user agrees to be a lady or a gentleman and to leave it alone.

If, later on, we find out that we need the class elsewhere (for instance, because we want to use `EuropeanBasketPathPricer` as a control-variate path pricer for another engine) we can move it to a header file—or, if it's already in one, to the main `QuantLib` namespace—and make it accessible to everybody.

\* \* \*

After all this, you might still have a question: how generic is this engine, really? Well, somewhat less than I'd like. It is generic in the sense that you can plug in any process for  $N$  asset prices, and it will work. It can even do more exotic stuff, such as quanto effects: if you go the traditional way and model the effect as a correction for the drift of the assets, you can write (or better yet, decorate) your process so that its `drift` or `evolve` method takes into account the correction, and you'll be able to use it without changing the engine code.

However, if you want to use a process that models other stochastic variables besides the asset prices (say, a multi-asset Heston process) you're likely to get into some trouble. The problem is that you'll end up in the `operator()` of the path pricer with  $N$  quantities and  $2N$  paths, without any indication of which are for the prices and which for the volatilities. How should they be combined to get the correct basket price? Of course, one can write a specific pricer for any particular process; but I'd like to provide something more reusable.

One simple solution is for the process and the pricer to share some coding convention. For instance, if you arrange the process so that the first  $N$  paths are those of the prices and the last  $M$  are those of any other variables, the pricer I've shown will work; note that it loops over the number of quantities, not the size of the process. However, this leaves one wide open to errors that will go undetected if a process doesn't conform to the convention.

Another possibility that comes to mind is to decorate the process with a layer that would show the asset prices and hide the extra variables. The decorator would appear as a process requiring the same number of random variates as the original, but exposing less stochastic variables. It would have to store some state in order to keep track of the hidden variables (which wouldn't be passed to its methods): its `evolve` method, to name one, would have to take the current prices, add the extra variables stored in the instance, call `evolve` in the original process, store the new values of the extra variables so that they're available for next call, and return only the new asset prices.

However, I'm not a fan of this solution. It would only work when the decorated methods are called in the expected order, and would break otherwise; for instance, if you called a method twice with the exact same arguments (starting prices and random variates) you would get different return, due to the changed internal state. The long way to express this is that the methods of the decorated process would lose referential transparency. The short way is that its interface would be a lie.

A more promising possibility (albeit one that requires some changes to the framework) would be to let the path generator do the filtering, with some help from the process. If the process could provide a method returning some kind of mask—say, a list of the indices of the assets into the whole set of variables, or a vector of booleans where the  $i$ -th element would be set to `true` if the  $i$ -th path corresponds to an asset price—then the generator could use it to drive the evolution of the variables correctly, and at the same time write in the multi-path only the asset prices. To make the whole thing backward-compatible, the mask would be passed to the generator only optionally, and the base `StochasticProcess` class would have a default implementation of the method returning an empty mask. If either way no mask were provided, the generator would revert to the current behavior. If needed, the method could be generalized to return other masks besides that for the prices; for instance, a variance swap would be interested in the volatility but not the price.

Finally, note that some assumptions are built right into the engine and cannot be relaxed without rewriting it. For instance, we're assuming that the discount factor for the exercise date is deterministic. If you wrote a process that models stochastic interest rates as well as asset prices, and therefore wanted to discount on each path according to the realization of the rates on that path, you'd have to write a custom engine and path pricer to be used with your specific process. Fortunately, that would be the only things you'd have to write; you'd still be able to reuse the other pieces of the framework.

# 7. The tree framework

Together with Monte Carlo simulations, trees are among the most commonly used tools in quantitative finance. As usual, the dual challenge for a framework is to implement a number of reusable (and possibly composable) pieces and to provide customization hooks for injecting new behavior. The QuantLib tree framework has gone through a few revisions, and the current version is a combination of object-oriented and generic programming that does the job without losing too much performance in the process.

## 7.1 The Lattice and DiscretizedAsset classes

The two main classes of the framework are the `Lattice` and `DiscretizedAsset` classes, shown in the next two listings.

Interface of the `Lattice` class.

---

```
class Lattice {
public:
    Lattice(const TimeGrid& timeGrid) : t_(timeGrid) {}
    virtual ~Lattice() {}

    const TimeGrid& timeGrid() const { return t_; }

    virtual void initialize(DiscretizedAsset&,
                           Time time) const = 0;
    virtual void rollback(DiscretizedAsset&,
                           Time to) const = 0;
    virtual void partialRollback(DiscretizedAsset&,
                               Time to) const = 0;
    virtual Real presentValue(DiscretizedAsset&) const = 0;

    virtual Disposable<Array> grid(Time) const = 0;
protected:
    TimeGrid t_;
};
```

---

---

**Implementation of the `DiscretizedAsset` class.**

---

```
class DiscretizedAsset {
public:
    DiscretizedAsset()
        : latestPreAdjustment_(QL_MAX_REAL),
          latestPostAdjustment_(QL_MAX_REAL) {}
    virtual ~DiscretizedAsset() {}

    Time time() const { return time_; }
    Time& time() { return time_; }
    const Array& values() const { return values_; }
    Array& values() { return values_; }
    const shared_ptr<Lattice>& method() const {
        return method_;
    }

    void initialize(const shared_ptr<Lattice>&,
                    Time t);
    void rollback(Time to);
    void partialRollback(Time to);
    Real presentValue();

    virtual void reset(Size size) = 0;
    void preAdjustValues();
    void postAdjustValues();
    void adjustValues();

    virtual std::vector<Time> mandatoryTimes() const = 0;

protected:
    bool isOnTime(Time t) const;
    virtual void preAdjustValuesImpl() {}
    virtual void postAdjustValuesImpl() {}

    Time time_;
    Time latestPreAdjustment_, latestPostAdjustment_;
    Array values_;

private:
    shared_ptr<Lattice> method_;
};

void DiscretizedAsset::initialize(
```

```
        const shared_ptr<Lattice>& method,
        Time t) {
    method_ = method;
    method_->initialize(*this, t);
}

void DiscretizedAsset::rollback(Time to) {
    method_->rollback(*this, to);
}

void DiscretizedAsset::partialRollback(Time to) {
    method_->partialRollback(*this, to);
}

Real DiscretizedAsset::presentValue() {
    return method_->presentValue(*this);
}

void DiscretizedAsset::preAdjustValues() {
    if (!close_enough(time(), latestPreAdjustment_)) {
        preAdjustValuesImpl();
        latestPreAdjustment_ = time();
    }
}

void DiscretizedAsset::postAdjustValues() {
    if (!close_enough(time(), latestPostAdjustment_)) {
        postAdjustValuesImpl();
        latestPostAdjustment_ = time();
    }
}

void DiscretizedAsset::adjustValues() {
    preAdjustValues();
    postAdjustValues();
}

bool DiscretizedAsset::isOnTime(Time t) const {
    const TimeGrid& grid = method()->timeGrid();
    return close_enough(grid[grid.index(t)], time());
}
```

---

In our intentions, the `Lattice` class was to model the generic concept of a discrete lattice, which might have been a tree as well as a finite-difference grid. This never happened; the finite-difference framework went its separate way and is unlikely to come back any time soon. However, the initial design helped keeping the `Lattice` class clean: to this day, it contains almost no implementation details and is not tied to trees.

Its constructor takes a `TimeGrid` instance and stores it (its only concession to implementation inheritance, together with an inspector that returns the time grid). All other methods are pure virtual. The `initialize` method must set up a discretized asset so that it can be put on the lattice at a given time;<sup>1</sup> the `rollback` and `partialRollback` methods roll the asset backwards in time on the lattice down to the desired time (with a difference I'll explain later); and the `presentValue` method returns what its name says.

Finally, the `grid` method returns the values of the discretized quantity underlying the lattice. This is a bit of a smell. The information was required in other parts of the library, and we didn't have any better solution. However, this method has obvious shortcomings. On the one hand, it constrains the return type, which either leaks implementation or forces a type conversion; and on the other hand, it simply makes no sense when the lattice has more than one factor, since the grid should be a matrix or a cube in that case. In fact, two-factor lattices implement it by having it throw an exception. All in all, this method could use some serious improvement in future versions of the library.

The `DiscretizedAsset` class is the base class for the other side of the tree framework—the Costello to `Lattice`'s Abbott, as it were. It models an asset that can be priced on a lattice: it works hand in hand with the `Lattice` class to provide generic behavior, and has hooks that derived classes can use to add behavior specific to the instrument they implement.

As can be seen from the listing, it's not nearly as abstract as the `Lattice` class. Most of its methods are concrete, with a few virtual ones that use the Template Method pattern to inject behavior.

Its constructor takes no argument, but initializes a couple of internal variables. The main inspectors return the data comprising its state, namely, the time  $t$  of the lattice nodes currently occupied by the asset and its values on the same nodes; both inspectors give both read and write access to the data to allow the lattice implementation to modify them. A read-only inspector returns the lattice on which the asset is being priced.

The next bunch of methods implements the common behavior that is inherited by derived classes and provide the interface to be called by client code. The body of a tree-based engine will usually contain something like the following after instantiating the tree and the discretized asset:

```
asset.initialize(lattice, T);
asset.rollback(t0);
results_.value = asset.presentValue();
```

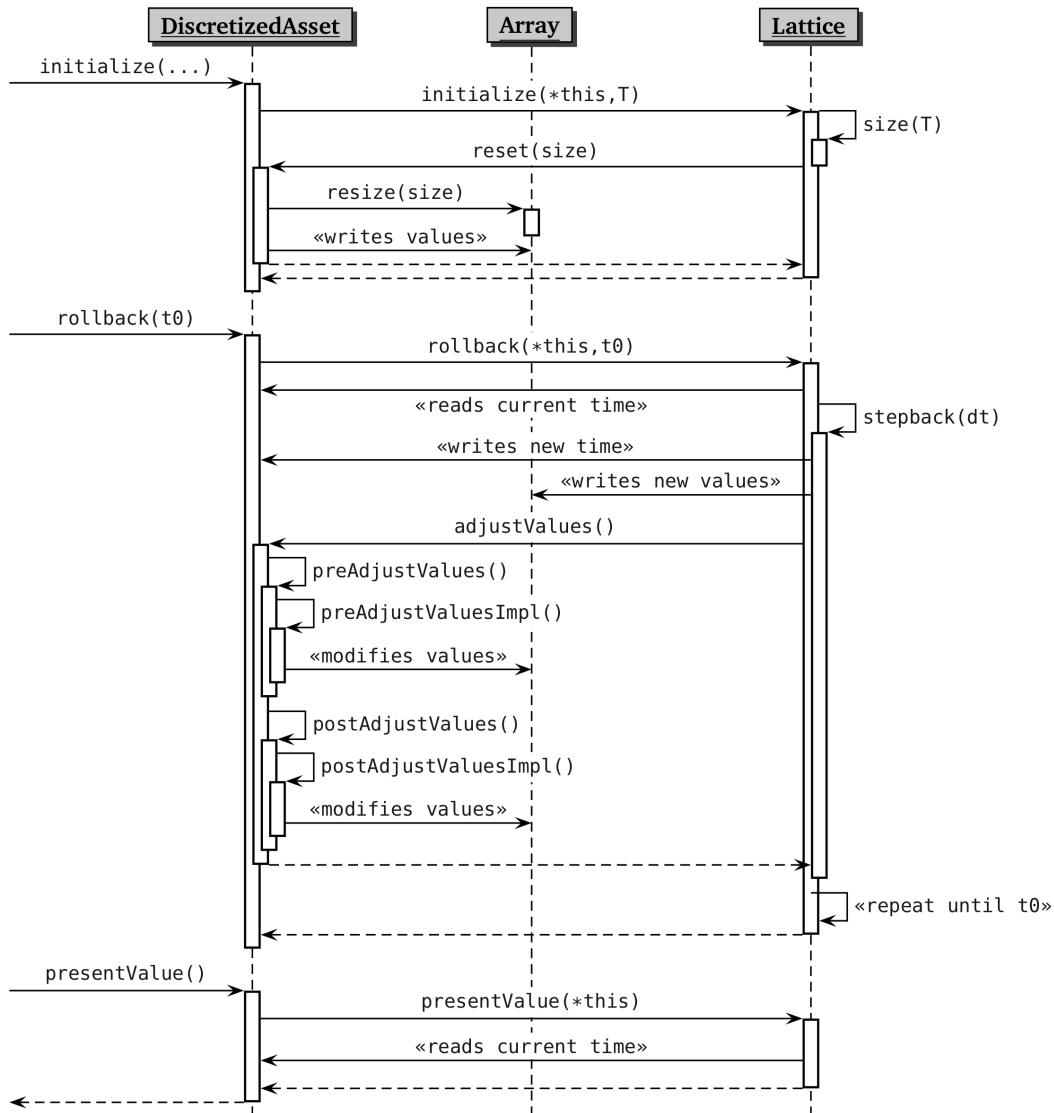
The `initialize` method stores the lattice that will be used for pricing (whose time grid must include the set of times returned by the `mandatoryTimes` method) and sets the initial values of the asset, or

---

<sup>1</sup>I do realize this is mostly hand-waving right now. It will become clear as soon as we get to a concrete lattice.

rather its final values given that the time  $T$  passed at initialization is the maturity time; the `rollback` method rolls the asset backwards on the lattice until the time  $t_0$ ; and the `presentValue` method extracts the value of the asset as a single number.

The three method calls above seem simple, but their implementation triggers a good deal of behavior in both the asset and the lattice and involves most of the other methods of `DiscretizedAsset` as well as those of `Lattice`. The interplay between the two classes is not nearly as funny as *Who's on First*, but it's almost as complex to follow; thus, you might want to refer to the sequence diagram shown in the figure that follows.



Sequence diagram of the interplay between the `DiscretizedAsset` and `Lattice` classes.

The `initialize` method sets the calculation up by placing the asset on the lattice at its maturity

time (the asset's, I mean) and preparing it for rollback. This means that on the one hand, the vector holding the asset values on each lattice node must be dimensioned correctly; and on the other hand, that it must be filled with the correct values. Like most of the `DiscretizedAsset` methods, `initialize` does this by delegating part of the actual work to the passed lattice; after storing it in the corresponding data member, it calls the lattice's `initialize` method passing the maturity time and the asset itself.

Now, the `Lattice` class doesn't implement `initialize`, which is left as purely virtual; but any sensible implementation in derived classes will do the dance shown in the sequence diagram. It might perform some housekeeping step of its own, not shown here; but first, it will determine the size of the lattice at the maturity time (that is, the number of nodes) probably by calling a corresponding method; then, it will store the time into the asset<sup>2</sup> and pass the size to the asset's `reset` method. The latter, implemented in derived classes, will resize the vector of values accordingly and fill it with instrument-specific values (for instance, a bond might store its redemption and the final coupon, if any).

Next is the `rollback` method. Again, it calls the corresponding method in the `Lattice` class, which performs the heavy-machinery work of stepping through the tree (or whatever kind of lattice it models). It reads the current time from the asset, so it knows on what nodes it is currently sitting; then, a close interplay begins.

The point is, the lattice can't simply roll the asset back to the desired time, since there might be all kind of events occurring: coupon payments, exercises, you name it. Therefore, it rolls it back only one short step, modifies the asset values accordingly—which includes both combining nodes and discounting—and then pauses to ask the asset if there's anything that needs to be done; that is, to call the asset's `adjustValues` method.

The adjustment is done in two steps, calling first the `preAdjustValues` and then the `postAdjustValue` method. This is done so that other assets have a chance to perform their own adjustment between the two; I'll show an example of this later on. Each of the two methods performs a bit of housekeeping<sup>3</sup> and then calls a virtual method (`preAdjustValuesImpl` or `postAdjustValueImpl`, respectively) which makes the instrument-specific adjustments.

When this is done, the ball is back in the lattice's field. The lattice rolls back another short step, and the whole thing repeats again and again until the assets reaches the required time.

Finally, the `presentValue` method returns, well, the present value of the asset. It is meant to be called by client code after rolling the asset back to the earliest interesting time—that is, to the time for which further rolling back to today's date only involves discounting and not adjustments of any kind: e.g., it might be the earliest exercise time of an option or the first coupon time of a bond. As usual, the asset delegates the calculation by passing itself to the lattice's `presentValue` method; a given lattice implementation might roll the asset all the way back to the present time (if it isn't already there, of course) and read its value from the root node, whereas another lattice might be

---

<sup>2</sup>As you remember, the asset provides read/write access to its state.

<sup>3</sup>Namely, they store the time of the latest adjustment so that the asset is not adjusted twice at the same time; this might happen when composing assets, and would obviously wreak havoc on the results.

able to take a shortcut and calculate the present value as a function of the values on the nodes at the current time.

As usual, there's room for improvement. Some of the steps are left as an implicit requirement; for instance, the fact that the lattice's `initialize` method should call the asset's `reset` method, or that `reset` must resize the array stored in the asset.<sup>4</sup> However, this is what we have at this time. Let's move on and build some assets.

### 7.1.1 Example: discretized bonds

Vanilla bonds (zero-coupon, fixed-rate and floating-rate) are simple enough to work as first examples, but at the same time provide a range of features large enough for me to make a few points.

Well, maybe not the zero-coupon bond. It's probably the simplest possible asset (bar one with no payoff) so it's not that interesting on its own; however, it's going to be useful as a helper class in the examples that follow, since it provides a way to estimate discount factors on the lattice.

Its implementation is shown in the next listing. It defines no mandatory times (because it will be happy to be initialized at any time you choose), it performs no adjustments (because nothing ever happens during its life), and its `reset` method fills each value in the array with one unit of currency. Thus, if you initialize an instance of his class at a time  $T$  and roll it back on a lattice until an earlier time  $t$ , the array values will then equal the discount factors from  $T$  to  $t$  as seen from the corresponding lattice nodes.

Implementation of the `DiscretizedDiscountBond` class.

---

```
class DiscretizedDiscountBond : public DiscretizedAsset {
public:
    DiscretizedDiscountBond() {}
    void reset(Size size) {
        values_ = Array(size, 1.0);
    }
    std::vector<Time> mandatoryTimes() const {
        return std::vector<Time>();
    }
};
```

---

\* \* \*

Things start to get more interesting when we turn to fixed-rate bonds. QuantLib doesn't provide discretized fixed-rate bonds (they're not needed, since such bonds are priced by discounting); the following listing shows a simple implementation which works here as an example.

---

<sup>4</sup>In hindsight, the array should be resized in the lattice's `initialize` method before calling `reset`. This would minimize repetition, since we'll write many more assets than lattices.

Implementation of the `DiscretizedFixedRateBond` class.

---

```

class DiscretizedFixedRateBond : public DiscretizedAsset {
public:
    DiscretizedFixedRateBond(const vector<Time>& paymentTimes,
                           const vector<Real>& coupons,
                           Real redemption)
        : paymentTimes_(paymentTimes), coupons_(coupons),
          redemption_(redemption) {}

    vector<Time> mandatoryTimes() const {
        return paymentTimes_;
    }
    void reset(Size size) {
        values_ = Array(size, redemption_);
        adjustValues();
    }
private:
    void postAdjustValuesImpl() {
        for (Size i=0; i<paymentTimes_.size(); i++) {
            Time t = paymentTimes_[i];
            if (t >= 0.0 && isOnTime(t)) {
                addCoupon(i);
            }
        }
    }
    void addCoupon(Size i) {
        values_ += coupons_[i];
    }

    vector<Time> paymentTimes_;
    vector<Real> coupons_;
    Real redemption_;
};
```

---

The constructor takes and stores a vector of times holding the payment schedule, the vector of the coupon amounts, and the amount of the redemption. Note that the type of the arguments is different from what the corresponding instrument class is likely to store (say, a vector of `CashFlow` instances); this implies that the pricing engine will have to perform some conversion work before instantiating the discretized asset.

The presence of a payment schedule implies that, unlike the zero-coupon bond above, this bond cannot be instantiated at just any time; and in fact, this class implements the `mandatoryTimes`

method by returning the vector of the payment times. Rollback on the lattice will have to stop at each such time, and initialization will have to be performed at the maturity time of the bond, i.e., the latest of the returned times. When one does so, the `reset` method will fill each value in the array with the redemption amount and then call the `adjustValues` method, which will take care of the final coupon.

In order to enable `adjustValues` to do so, this class overrides the virtual `postAdjustValuesImpl` method. (Why the *post-* version of the method and not the *pre-*, you say? Bear with me a few more pages: I need a bit more context to explain. All will become clear.) The method loops over the payment times and checks, by means of the probably poorly named `isOnTime` method, whether any of them equals the current asset time. If this is the case, we add the corresponding coupon amount to the asset values. For readability, the actual work is factored out in the `addCoupon` method. The coupon amounts will be automatically discounted as the asset is rolled back on the lattice.

\* \* \*

Finally, let's turn to the floating-rate bond, shown in the next listing; this, too, is a simplified implementation.

Implementation of the `DiscretizedFloatingRateBond` class.

---

```

class DiscretizedFloatingRateBond : public DiscretizedAsset {
public:
    DiscretizedFloatingRateBond(const vector<Time>& paymentTimes,
                               const vector<Time>& fixingTimes,
                               Real notional)
        : paymentTimes_(paymentTimes), fixingTimes_(fixingTimes),
          notional_(notional) {}

    vector<Time> mandatoryTimes() const {
        vector<Time> times = paymentTimes_;
        std::copy(fixingTimes_.begin(), fixingTimes_.end(),
                  back_inserter(times));
        return times;
    }
    void reset(Size size) {
        values_ = Array(size, notional_);
        adjustValues();
    }
private:
    void preAdjustValuesImpl() {
        for (Size i=0; i<fixingTimes_.size(); i++) {
            Time t = fixingTimes_[i];

```

```

    if (t >= 0.0 && isOnTime(t)) {
        addCoupon(i);
    }
}

void addCoupon(Size i) {
    DiscretizedDiscountBond bond;
    bond.initialize(method(), paymentTimes_[i]);
    bond.rollback(time_);

    for (Size j=0; j<values_.size(); j++) {
        values_[j] += notional_ * (1.0 - bond.values()[j]);
    }
}

vector<Time> paymentTimes_;
vector<Time> fixingTimes_;
Real notional_;
};
```

---

The constructor takes and stores the vector of payment times, the vector of fixing times, and the notional of the bond; there are no coupon amounts, since they will be estimated during the calculation. For simplicity of implementation, we'll assume that the accrual time for the  $i$ -th coupon equals the time between its fixing time and its payment time.

The `mandatoryTimes` method returns the union of payment times and fixing times, since we'll need to work on both during the calculations. The `reset` method is similar to the one for fixed-rate bonds, and fills the array with the redemption value (which equals the notional of the bond) before calling `adjustValues`.

Adjustment is performed in the overridden `preAdjustValuesImpl` method. (Yes, the *pre-* version. Patience.) It loops over the fixing times, checks whether any of them equals the current time, and if so calls the `addCoupon` method.

Now, like the late Etta James in one of her hits, you'd be justified in shouting "Stop the wedding". Of course the coupon should be added at the payment date, right? Well, yes; but the problem is that we're going backwards in time. In general, at the payment date we don't have enough information to add the coupon; it can only be estimated based on the value of the rate at an earlier time that we haven't yet reached. Therefore, we have to keep rolling back on the lattice until we get to the fixing date, at which point we can calculate the coupon amount and add it to the bond. In this case, we'll have to take care ourselves of discounting from the payment date, since we passed that point already.

That's exactly what the `addCoupon` method does. First of all, it instantiates a discount bond at the payment time  $T$  and rolls it back to the current time, i.e., the fixing date  $t$ , so that its value equal at

the  $j$ -th node the discount factors  $D_j$  between  $t$  and  $T$ . From those, we could estimate the floating rates  $r_j$  (since it must hold that  $1 + r_j(T - t) = 1/D_j$ ) and then the coupon amounts; but with a bit of algebra, we can find a simpler calculation. The coupon amount  $C_j$  is given by  $Nr_j(T - t)$ , with  $N$  being the notional; and since the relation above tells us that  $r_j(T - t) = 1/D_j - 1$ , we can substitute that to find that  $C_j = N(1/D_j - 1)$ . Now, remember that we already rolled back to the fixing date, so if we add the amount here we also have to discount it because it won't be rolled back from the payment date. This means that we'll have to multiply it by  $D_j$ , and thus the amount to be added to the  $j$ -th value in the array is  $C_j = N(1/D_j - 1)D_j = N(1 - D_j)$ . The final expression is the one that can be seen in the implementation of `addCoupon`.

As you probably noted, the above hinges on the assumption that the accrual time equals the time between payment and fixing time. If this were not the case, the calculation would no longer simplify and we'd have to change the implementation; for instance, we might instantiate a first discount bond at the accrual end date to estimate the floating rate and the coupon amount, and a second one at the payment date to calculate the discount factors to be used when adding the coupon amount to the bond value. Of course, the increased accuracy would cause the performance to degrade since `addCoupon` would roll back two bonds, instead of one. You can choose either implementation based on your requirements.

### 7.1.2 Example: discretized option

Sorry to have kept you waiting, folks. Here is where I finally explain the pre- vs post-adjustment choice, after the previous example helped me put my ducks in a row. I'll do so by showing an example of an asset class (the `DiscretizedOption` class, shown in the next listing) that can be used to wrap an underlying asset and obtain an option to enter the same: for instance, it could take a swap and yield a swaption. The implementation shown here is a slightly simplified version of the one provided by QuantLib, since it assumes a Bermudan exercise (or European, if one passes a single exercise time). Like the implementation in the library, it also assumes that there's no premium to pay in order to enter the underlying deal.

Implementation of the `DiscretizedOption` class.

---

```
class DiscretizedOption : public DiscretizedAsset {
public:
    DiscretizedOption(
        const shared_ptr<DiscretizedAsset>& underlying,
        const vector<Time>& exerciseTimes)
        : underlying_(underlying), exerciseTimes_(exerciseTimes) {}

    vector<Time> mandatoryTimes() const {
        vector<Time> times = underlying_->mandatoryTimes();
        for (Size i=0; i<exerciseTimes_.size(); ++i)
            if (exerciseTimes_[i] >= 0.0)
                times.push_back(exerciseTimes_[i]);
    }
}
```

```

    return times;
}
void reset(Size size) {
    QL_REQUIRE(method() == underlying_->method(),
               "option and underlying were initialized on "
               "different lattices");
    values_ = Array(size, 0.0);
    adjustValues();
}
private:
void postAdjustValuesImpl() {
    underlying_->partialRollback(time());
    underlying_->preAdjustValues();
    for (Size i=0; i<exerciseTimes_.size(); i++) {
        Time t = exerciseTimes_[i];
        if (t >= 0.0 && isOnTime(t))
            applyExerciseCondition();
    }
    underlying_->postAdjustValues();
}
void DiscretizedOption::applyExerciseCondition() {
    for (Size i=0; i<values_.size(); i++)
        values_[i] = std::max(underlying_->values()[i],
                             values_[i]);
}

shared_ptr<DiscretizedAsset> underlying_;
vector<Time> exerciseTimes_;
};
```

---

Onwards. The constructor takes and, as usual, stores the underlying asset and the exercise times; nothing to write much about.

The `mandatoryTimes` method takes the vector of times required by the underlying and adds the option's exercise times. Of course, this is done so that both the underlying and the option can be priced on the same lattice; the sequence of operations to get the option price will be something like:

```

underlying = shared_ptr<DiscretizedAsset>(...);
option = shared_ptr<DiscretizedAsset>(
    new DiscretizedOption(underlying, exerciseTimes));
grid = TimeGrid(..., option->mandatoryTimes());
lattice = shared_ptr<Lattice>(new SomeLattice(..., grid, ...));
underlying->initialize(lattice, T1);
option->initialize(lattice, T2);
option->rollback(t0);
NPV = option->presentValue();

```

in which, first, we instantiate both underlying and option and retrieve the mandatory times from the latter; then, we create the lattice and initialize both assets (usually at different times, e.g., the maturity date for a swap and the latest exercise date for the corresponding swaption); and finally, we roll back the option and get its price. As we'll see in a minute, the option also takes care of rolling the underlying back as needed.

Back to the class implementation. The `reset` method performs the sanity check that underlying and option were initialized on the same lattice, fills the values with zeroes (what you end up with if you don't exercise), and then calls `adjustValues` to take care of a possible exercise.

Which brings us to the crux of the matter, i.e., the `postAdjustValuesImpl` method. The idea is that, if we're on an exercise time, we need to check whether keeping the option is worth more than entering the underlying asset. To do so, we roll the underlying asset back to the current time, compare values at each node, and set the option value to the maximum of the two; this latest part is abstracted out in the `applyExerciseCondition` method.

The tricky part of the problem is that the underlying might need to perform an adjustment of its own when rolled back to the current time. Should this be done before or after the option looks at the underlying values?

It depends on the particular adjustment. Let's look at the bonds in the previous example. If the underlying is a discretized fixed-rate bond, and if the current time is one of its payment times, it needs to adjust its values by adding a coupon. This coupon, though, is being paid now and thus is no longer part of the asset if we exercise and enter it. Therefore, the decision to exercise must be based on the bond value without the coupon; i.e., we must call the `applyExerciseCondition` method before adjusting the underlying.

The discretized floating-rate bond is another story. It adjusts the values if the current time is one of its fixing times; but in this case the corresponding coupon is just starting and will be paid at the end of the period, and so must be added to the bond value before we decide about exercise. Thus, the conclusion is the opposite: we must call `applyExerciseCondition` after adjusting the underlying.

What should the option do? It can't distinguish between the two cases, since it doesn't know the specific behavior of the asset it was passed; therefore, it lets the underlying itself sort it out. First, it rolls the underlying back to the current time, but without performing the final adjustment (that's what the `partialRollback` method does); instead, it calls the underlying's `preAdjustValues`

method. Then, if we're on an exercise time, it performs its own adjustment; and finally, it calls the underlying's `postAdjustValues` method.

This is the reason the `DiscretizedAsset` class has both a `preAdjustValues` and a `postAdjustValues` method; they're there so that, in case of asset composition, the underlying can choose on which side of the fence to be when some other adjustment (such as an exercise) is performed at the same time. In the case of our previous example, the fixed-rate bond will add its coupon in `postAdjustValues` and have it excluded from the future bond value, while the floating-rate bond will add its coupon in `preAdjustValues` and have it included.

Unfortunately, this solution is not very robust. For instance, if the exercise dates were a week or two before the coupon dates (as is often the case) the option would break for fixed-rate coupons, since it would have no way to stop them from being added before the adjustment. The problem can be solved: in the library, this is done for discretized interest-rate swaps by adding fixed-rate coupons on their start dates, much in the same way as floating-rate coupons. Another way to fix the issue would be to roll the underlying back only to the date when it's actually entered, then to make a copy of it and roll the copy back to the exercise date without performing any adjustment. Both solutions are somewhat clumsy at this time; it would be better if QuantLib provided some means to do it more naturally.

## 7.2 Trees and tree-based lattices

In order to use the assets I just described, we need a lattice. The first question we have to face is at the very beginning of its design: should the tree itself be the lattice, or should we have a separate lattice class using a tree?

Even if the first alternative can be argued (a tree can very well be seen as a kind of lattice, so an *is-a* relationship would be justified), the library implements the second. One reason is that having separate classes for the lattice and tree allows us to separate their concerns: the tree has the responsibility for maintaining the structure and the probability transitions between nodes, and the lattice uses the tree structure to roll the asset back and adds some additional features such as discounting. Another is that, as we'll see, a lattice can use more than one tree.

### 7.2.1 The Tree class template

As I just wrote, a tree provides information about its structure. It can be described by the number of nodes at each level (each level corresponding to a different time), the nodes that can be reached from each other node at the previous level, and the probabilities for each such transition.

The interface chosen to represent this information (shown in the listing that follows) was an index-based one. Nodes were not represented as instances of a `Node` class or structure; instead, the tree was described—not implemented, mind you—as a series of vectors of increasing size, and the nodes were identified by their indexes into the vectors. This gives more flexibility in implementing different trees; one tree class might store the actual vectors of nodes, while another might just calculate the

required indexes as needed. In the library, trinomial trees are implemented in the first way and binomial trees in the second. I'll describe both later on.

Original interface of the `Tree` class.

---

```
class Tree {
public:
    Tree(Size columns);
    virtual ~Tree();
    Size columns() const;
    virtual Size size(Size i) const = 0;
    virtual Real underlying(Size i, Size j) const = 0;
    virtual Size branches() const = 0;
    virtual Size descendant(Size i, Size j,
                           Size branch) const = 0;
    virtual Real probability(Size i, Size index,
                           Size branch) const = 0;
};
```

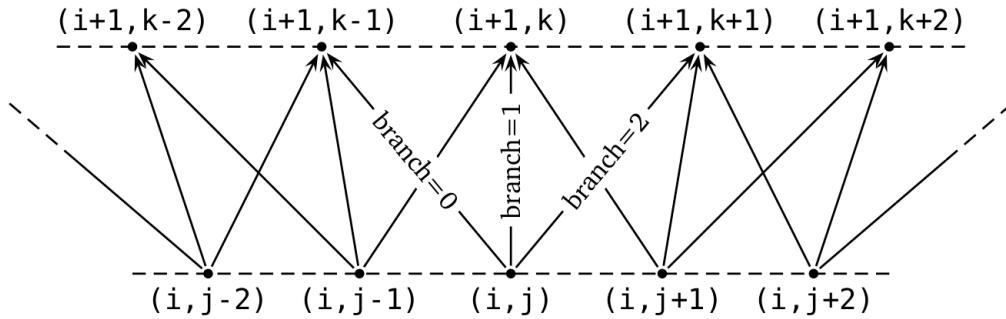
---

The methods shown in the interface fully describe the tree structure. The `columns` method returns the number of levels in the tree; evidently, the original designer visualized the time axis as going from left to right, with the tree growing in the same direction. If we were to change the interface, I'd probably go for a more neutral name, such as `size`.

The `size` name was actually taken for another method, which returns the number of nodes at the `i`-th level of the tree; the `underlying` method takes two indexes `i` and `j` and returns the value of the variable underlying the tree at the `j`-th node of the `i`-th level.

The remaining methods deal with branching. The `branches` method returns the number of branches for each node (2 for binomial trees, 3 for trinomial ones and so on). It takes no arguments, which means that we're assuming such a number to be constant; we'll have no trees with a variable number of branches. The `descendant` identifies the nodes towards which a given node is branching: it takes the index `i` of the level, the index `j` of the node and the index of a branch, and returns the index of the corresponding node on level `i+1` as exemplified in the figure below. Finally, the `probability` method takes the same arguments and returns the probability to follow a given branch.

$\text{descendant}(i, j, 0) \rightarrow k-1$	$\text{descendant}(i, j-2, 1) \rightarrow k-2$
$\text{descendant}(i, j, 1) \rightarrow k$	$\text{descendant}(i, j-1, 0) \rightarrow k-2$
$\text{descendant}(i, j, 2) \rightarrow k+1$	$\text{descendant}(i, j+1, 2) \rightarrow k+2$



Schematics of a sample tree.

The original implementation actually sported a base `Tree` class as shown in the previous listing. It worked, but it committed what amounts to a mortal sin in numerical code: it declared as virtual some methods (such as `descendant` or `probability`) that were meant to be called thousands of times inside tight nested loops.<sup>5</sup> After a while, we started to notice that small glaciers were disappearing in the time it took us to calibrate a short-rate model.

Eventually, this led to a reimplementation. We used the Curiously Recurring Template Pattern (CRTP) to replace run-time with compile-time polymorphism (see the aside [at the end of this section](#) if you need a refresher) which led to the `Tree` class template shown in the listing below. The virtual methods are gone, and only the non-virtual `columns` method remains; the machinery for the pattern is provided by the `CuriouslyRecurringTemplate` class template, which avoids repetition of boilerplate code by implementing the `const` and non-`const` versions of the `impl` method used in the pattern.

Implementation of the `Tree` class template.

---

```
template <class T>
class Tree : public CuriouslyRecurringTemplate<T> {
public:
    Tree(Size columns) : columns_(columns) {}
    Size columns() const { return columns_; }
private:
    Size columns_;
};
```

---

This was not a redesign; the existing class hierarchy was left unchanged. Switching to CRTP yielded

<sup>5</sup>If you're wondering why this is bad, see for instance ([Veldhuizen, 2000](#)) for a short explanation—along with a lot more information on using C++ for numerical work.

slightly more complex code, as you'll see when I describe binomial and trinomial trees. However, it paid off in speed: with the new implementation, we saw the performance increase up to 10x. Seeing the results, I didn't stop to ask myself many question (yes, I'm the guilty party here) and I missed an opportunity to simplify the code. With a cooler head, I might have noticed that the base tree classes don't call methods defined in derived classes, so I could have simplified the code further by dropping CRTP for simple templates.<sup>6</sup> I don't know if I can deprecate my way out of this.

## Aside: curioser and curioser.

The Curiously Recurring Template Pattern (CRTP) was named and popularized by James Coplien in ([Coplien, 1996](#)) but it predates his description. It is a way to achieve a static version of the Template Method pattern (see ([Gamma et al, 1995](#)), of course), in which a base class implements common behavior and provides hooks for customization by calling methods that will be implemented in derived classes.

The idea is the following:

```
template <T>
class Base {
    public:
        T& impl() {
            return static_cast<T&>(*this);
        }
        void foo() {
            ...
            this->impl().bar();
            ...
        }
};

class Derived : public Base<Derived> {
    public:
        void bar() { ... }
};
```

Now if we instantiate a `Derived` and call its `foo` method, it will in turn call the correct `bar` implementation: that is, the one defined in `Derived` (note that `Base` doesn't declare `bar` at all).

We can see how this work by working out the expansion made by the compiler. The `Derived` class doesn't define a `foo` method, so it inherits it from the its base class; that is, `Base<Derived>`. The `foo` method calls `impl`, which casts `*this` to a reference to the template argument `T`, i.e., `Derived` itself. We called the method on a `Derived` instance to begin with, so the cast succeeds and we're left with a reference to our object that has the correct type. At this point, the compiler resolves the call to `bar` on the reference to `Derived::bar` and thus the correct method is called. All this happens

---

<sup>6</sup>If a new branch of the hierarchy were to need CRTP, it could be added locally to that branch.

at compile time, and enables the compiler to inline the call to `bar` and perform any optimization it sees fit.

Why go through all this, and not just call `bar` from `foo` instead? Well, it just wouldn't work. If we didn't declare `bar` into `Base`, the compiler wouldn't know what method we're talking about; and if we did declare it, the compiler would always resolve the call to the `Base` version of the method. CRTP gives us a way out: we can define the method in the derived class and use the template argument to pass its type to the base class so that the static cast can close the circle.

## 7.2.2 Binomial and trinomial trees

As I mentioned earlier, a tree might implement its required interface by storing actual nodes or by just calculating their indexes on demand. The `BinomialTree` class template, shown in the listing below, takes the second approach. In a way, the nodes *are* the indexes; they don't exist as separate entities.

Implementation of the `BinomialTree` class template.

---

```
template <class T>
class BinomialTree : public Tree<T> {
public:
    enum Branches { branches = 2 };
    BinomialTree(
        const shared_ptr<StochasticProcess1D>& process,
        Time end, Size steps)
        : Tree<T>(steps+1) {
            x0_ = process->x0();
            dt_ = end/steps;
            driftPerStep_ = process->drift(0.0, x0_) * dt_;
    }
    Size size(Size i) const {
        return i+1;
    }
    Size descendant(Size, Size index, Size branch) const {
        return index + branch;
    }
protected:
    Real x0_, driftPerStep_;
    Time dt_;
};
```

---

The class template models a rather strict subset of the possible binomial trees, that is, recombining trees with constant drift and diffusion for the underlying variable<sup>7</sup> and with constant time steps.

Even with these constraints, there's quite a few different possibilities for the construction of the tree, so most of the implementation is left to derived classes. This class acts as a base and provides the common facilities; first of all, a `branches` enumeration whose value is defined to be 2 and which replaces the corresponding method in the old interface. Nowadays, we'd use a `static const Size` data member for the purpose; the enumeration is a historical artifact from an ancient time when compilers were less standard-compliant.

The class constructor takes a one-dimensional stochastic process providing the dynamics of the underlying, the end time of the tree (with the start time assumed to be 0, another implicit constraint), and the number of time steps. It calculates and stores a few simple quantities: the initial value of the underlying, obtained by calling the `x0` method of the process; the size of a time step, obtained by dividing the total time by the number of steps; and the amount of underlying drift per step, once again provided by the process (since the parameters are constant, the drift is calculated at  $t = 0$ ). The class declares corresponding data members for each of these quantities.

Unfortunately, there's no way to check that the given process is actually one with constant parameters. On the one hand, the library doesn't declare a specific type for that kind of process; it would be orthogonal to the declaration of different process classes, and it would lead to multiple inheritance—the bad kind—if one wanted to define, say, a constant-parameter Black-Scholes process (it would have to be both a constant-parameter process and a Black-Scholes process). Therefore, we can't enforce the property by restricting the type to be passed to the constructor. On the other hand, we can't write run-time tests for that: we might check a few sample points, but there's no guarantee that the parameters don't change elsewhere.

One option that remains is to document the behavior and trust the user not to abuse the class. Another would be for the constructor to forgo the process and take the constant parameters instead; however, this would give the user the duty to extract the parameters from the process at each constructor invocation. Yet another would be to separate model and process, as I mentioned [in section 6.1.2](#). As usual, we'll have to find some kind of balance.

The two last methods define the structure of the tree. The `size` method returns the number of nodes at the  $i$ -th level; the code assumes that there's a single node (the root of the tree) at level 0, which gives  $i + 1$  nodes at level  $i$  because of recombination. This seems reasonable enough, until we realize that we also assumed that the tree starts at  $t = 0$ . Put together, these two assumptions prevent us from implementing techniques such as, say, having three nodes at  $t = 0$  in order to calculate Delta and Gamma by finite differences. Luckily enough, this problem might be overcome without losing backward compatibility: if one were to try implementing it, the technique could be enabled by adding additional parameters to the constructors, thus allowing to relax the assumptions above.

Finally, the `descendant` method describes the links between the nodes. Due to the recombining structure of the tree, the node at index 0 on one level connects to the two nodes at index 0 and 1

---

<sup>7</sup>The library contains, in its “experimental” folder, an extension of this class to non-constant parameters. You're welcome to have a look at it and send us feedback. For illustration purposes, though, I'll stick to the constant version here.

on the next level, the node at index 1 to those at index 1 and 2, and in general the node at index  $i$  to those at index  $i$  and  $i + 1$ . Since the chosen branch is passed as an index (0 or 1 for a binomial tree) the method can add the index of the node to that of the branch to return the correct result. Neither index is range-checked, since the method will be called almost always inside a loop; checks would not only be costly (I haven't measured the effect, though, so I'll leave it at that) but actually redundant, since they will already be performed while setting up the loop.

The implementation of the `BinomialTree` class template misses the methods that map the dynamics of the underlying to the tree nodes: that is, the `underlying` and `probability` methods. They are left to derived classes, since there are several ways in which they can be written. In fact, there are families of ways: the first two class templates in the next listing are meant to be base classes for two such families.

A few classes derived from the `BinomialTree` class template.

---

```
template <class T>
class EqualProbabilitiesBinomialTree : public BinomialTree<T> {
public:
    EqualProbabilitiesBinomialTree(
        const shared_ptr<StochasticProcess1D>& process,
        Time end,
        Size steps)
        : BinomialTree<T>(process, end, steps) {}
    Real underlying(Size i, Size index) const {
        int j = 2*int(index) - int(i);
        return this->x0_*std::exp(i*this->driftPerStep_
            + j*this->up_);
    }
    Real probability(Size, Size, Size) const {
        return 0.5;
    }
protected:
    Real up_;
};

template <class T>
class EqualJumpsBinomialTree : public BinomialTree<T> {
public:
    EqualJumpsBinomialTree(
        const shared_ptr<StochasticProcess1D>& process,
        Time end,
        Size steps)
        : BinomialTree<T>(process, end, steps) {}
    Real underlying(Size i, Size index) const {
        int j = 2*int(index) - int(i);
```

```

    return this->x0_*std::exp(j*this->dx_);
}
Real probability(Size, Size, Size branch) const {
    return (branch == 1 ? pu_ : pd_);
}
protected:
Real dx_, pu_, pd_;
};

class JarrowRudd
: public EqualProbabilitiesBinomialTree<JarrowRudd> {
public:
    JarrowRudd(
        const shared_ptr<StochasticProcess1D>& process,
        Time end, Size steps, Real strike)
    : EqualProbabilitiesBinomialTree<JarrowRudd>(process,
                                                    end, steps) {
        up_ = process->stdDeviation(0.0, x0_, dt_);
    }
};

class CoxRossRubinstein
: public EqualJumpsBinomialTree<CoxRossRubinstein> {
public:
    CoxRossRubinstein(
        const shared_ptr<StochasticProcess1D>& process,
        Time end, Size steps, Real strike)
    : EqualJumpsBinomialTree<CoxRossRubinstein>(process,
                                                    end, steps) {
        dx_ = process->stdDeviation(0.0, x0_, dt_);
        pu_ = 0.5 + 0.5*driftPerStep_/dx_;
        pd_ = 1.0 - pu_;
    }
};

class Tian : public BinomialTree<Tian> {
public:
    Tian(const shared_ptr<StochasticProcess1D>& process,
          Time end, Size steps, Real strike)
    : BinomialTree<Tian>(process, end, steps) {
        // sets up_, down_, pu_, and pd_
    }
    Real underlying(Size i, Size index) const {

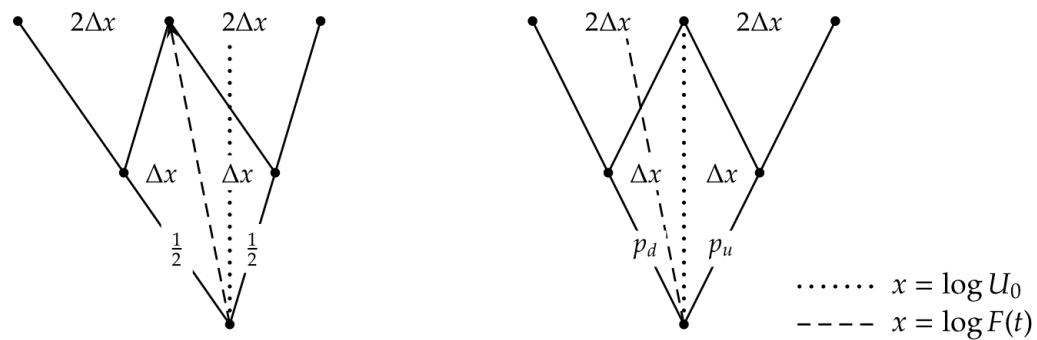
```

```

    return x0_ * std::pow(down_, Real(int(i)-int(index)))
        * std::pow(up_, Real(index));
}
Real probability(Size, Size, Size branch) const {
    return (branch == 1 ? pu_ : pd_);
}
protected:
    Real up_, down_, pu_, pd_;
};
```

---

The first, `EqualProbabilitiesBinomialTree`, can be used for those trees in which from each node there's the same probability to go to either branch. This determines the implementation of the `probability` method, which identically returns 0.5, but also sets constraints on the `underlying` method. Equal probability to go along each branch means that the spine of the tree (that is, the center nodes of each level) is the expected place to be; and in turn, this means that such nodes must be placed at the forward value of the underlying for the corresponding time (see the figure that follows for an illustration of the idea). The class defines an `underlying` method that implements the resulting formula for the logarithm of the underlying (you can tell that the whole thing was written with the Black-Scholes process in mind). A data member `up_` is defined and used to hold the half-displacement from the forward value per each move away from the center, represented by  $\Delta x$  in the figure; however, the constructor of the class does not give it a value, leaving this small freedom to derived classes.



Equal-probabilities vs equal-jumps binomial trees.

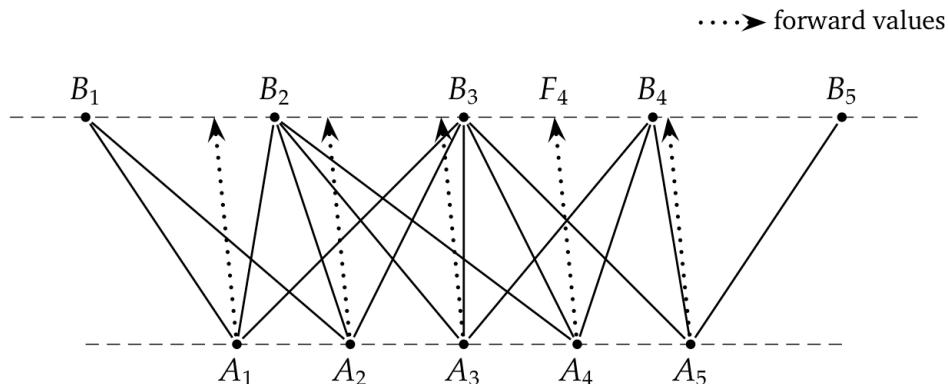
The second template, `EqualJumpsBinomialTree`, is used for those trees in which going to either branch implies an equal amount of movement in opposite directions for the logarithm of the underlying (again, see the previous figure); since one of the directions goes with the drift and the other against it, the probabilities are different. The template defines the `underlying` method accordingly, leaving to derived classes the task of filling the value of the `dx_` data member; and it also provides an implementation for the `probability` method that returns one of two values (`pu_` or

`pd_`) which must also be specified by derived classes.

Finally, the last three classes in the previous listing are examples of actual binomial trees, and as such are no longer class templates. The first implements the Jarrow-Rudd tree; it inherits from `EqualProbabilitiesBinomialTree` (note the application of CRTP) and its constructor sets the `up_` data member to the required value, namely, the standard deviation of the underlying distribution after one time step. The second implements the Cox-Ross-Rubinstein tree, inherits from `EqualJumpsBinomialTree`, and its constructor calculates the required probabilities as well as the displacement. The third one implements the Tian tree, and provides an example of a class that doesn't belong to either family; as such, it inherits from `BinomialTree` directly and provides all required methods.

\* \* \*

Trinomial trees are implemented as different beasts entirely, and have a lot more leeway in connecting nodes. The way they're built (which is explained in greater detail in [Brigo and Mercurio, 2006](#)) is sketched in the next figure: on each level in the tree, nodes are placed at an equal distance between them based on a center node with the same underlying value as the root of the tree; in the figure, the center nodes are  $A_3$  and  $B_3$ , placed on the same vertical line. As you can see, the distance can be different on each level.



Schematics of trinomial tree branching.

Once the nodes are in place, we build the links between them. Each node on a level corresponds, of course, to an underlying value  $x$  at the given time  $t$ . From each of them, the process gives us the expectation value for the next time conditional to starting from  $(x, t)$ ; this is represented by the dotted lines. For each forward value, we determine the node which is closest and use that node for the middle branch. For instance, let's look at the node  $A_4$  in the figure. The dynamics of the underlying gives us a forward value corresponding to the point  $F_4$  on the next level, which is closest to the node  $B_3$ . Therefore,  $A_4$  branches out to  $B_3$  in the middle and to its nearest siblings  $B_2$  on the left and  $B_4$  on the right.

As you see from the figure, it can well happen that two nodes on one level have forward values which are closest to the same node on the next level; see, for instance, nodes  $A_3$  and  $A_4$  going both to  $B_3$  in the middle. This means that, while it's guaranteed by construction that three branches start from each node, there's no telling beforehand how many branches go to a given node; here we range from  $B_5$  being the end node of just one branch to  $B_3$  being the end node of five. There is also no telling how many nodes we'll need at each level: in this case, five  $B$  nodes are enough to receive all the branches starting from five  $A$  nodes, but depending on the dynamics of the process we might have needed more nodes or fewer (you can try it: modify the distances in the figure and see what happens).

The logic I just described is implemented by QuantLib in the `TrinomialTree` class, shown in the following listing.

Sketch of the `TrinomialTree` class.

---

```
class TrinomialTree : public Tree<TrinomialTree> {
    class Branching;
public:
    enum Branches { branches = 3 };
    TrinomialTree(
        const shared_ptr<StochasticProcess1D>& process,
        const TimeGrid& timeGrid,
        bool isPositive = false);
    Real dx(Size i) const;
    const TimeGrid& timeGrid() const;
    Size size(Size i) const {
        return i==0 ? 1 : branchings_[i-1].size();
    }
    Size descendant(Size i, Size index, Size branch) const;
    Real probability(Size i, Size index, Size branch) const;
    Real underlying(Size i, Size index) const {
        return i==0 ? x0_;
        x0_ + (branchings_[i-1].jMin() + index)*dx(i);
    }
protected:
    std::vector<Branching> branchings_;
    Real x0_;
    std::vector<Real> dx_;
    TimeGrid timeGrid_;
};
```

---

Its constructor takes a one-dimensional process, a time grid specifying the times corresponding to each level of the tree (which don't need to be at regular intervals) and a boolean flag which, if set, constrains the underlying variable to be positive.

I'll get to the construction in a minute, but first I need to show how the information is stored; that is, I need to describe the inner `Branching` class, shown in the next listing. Each instance of the class stores the information for one level of the tree; e.g., one such instance could encode [the previous figure](#).

Implementation of the `TrinomialTree::Branching` inner class.

---

```

class TrinomialTree::Branching {
    public:
        Branching()
        : probs_(3), jMin_(QL_MAX_INTEGER), jMax_(QL_MIN_INTEGER) {}
        Size size() const {
            return jMax_ - jMin_ + 1;
        }
        Size descendant(Size i, Size branch) const {
            return k_[i] - jMin_ + branch - 1;
        }
        Real probability(Size i, Size branch) const {
            return probs_[branch][i];
        }
        Integer jMin() const;
        Integer jMax() const;
        void add(Integer k, Real p1, Real p2, Real p3) {
            k_.push_back(k);
            probs_[0].push_back(p1);
            probs_[1].push_back(p2);
            probs_[2].push_back(p3);

            jMin_ = std::min(jMin_, k-1);
            jMax_ = std::max(jMax_, k+1);
        }
    private:
        std::vector<Integer> k_;
        std::vector<std::vector<Real>> probs_;
        Integer jMin_, jMax_;
};
```

---

As I mentioned, nodes are placed based on a center node corresponding to the initial value of the underlying. That's the only available reference point, since we don't know how many nodes we'll use on either side. Therefore, the `Branching` class uses an index system that assigns the index  $j = 0$  to the center node and works outwards from there. For instance, on the lower level in the figure we'd have  $j = 0$  for  $A_3$ ,  $j = 1$  for  $A_4$ ,  $j = -1$  for  $A_2$  and so on; on the upper level, we'd start from  $j = 0$  for  $B_3$ . These indexes will have to be translated to those used by the tree interface, that start at 0 for the leftmost node ( $A_1$  in the figure).

To hold the tree information, the class declares as data members a vector of integers, storing for each lower-level node the index (in the branching system) of the corresponding mid-branch node on the upper level;<sup>8</sup> three vectors, declared for convenience of access as a vector of vectors, storing the probabilities for each of the three branches; and two integers storing the minimum and maximum node index used on the upper level (again, in the branching system).

Implementing the tree interface requires some care with the several indexes we need to juggle. For instance, let's go back to `TrinomialTree` and look at the `size` method. It must return the number of nodes at level  $i$ ; and since each branching holds information on the nodes of its upper level, the correct value must be retrieved from `branchings_[i-1]` (except for the case  $i = 0$ , for which the result is 1 by construction). To allow this, the `Branching` class provides a `size` method that returns the number of points on the upper level; since `jMin_` and `jMax_` store the indexes of the leftmost and rightmost node, respectively, the number to return is `jMax_ - jMin_ + 1`. In the figure, indexes go from  $-2$  to  $2$  (corresponding to  $B_1$  and  $B_5$ ) yielding  $5$  as the number of nodes.

The `descendant` and `probability` methods of the tree both call corresponding methods in the `Branching` class. The first returns the descendant of the  $i$ -th lower-level node on the given branch (specified as  $0$ ,  $1$  or  $2$  for the left, mid and right branch, respectively). To do so, it first retrieves the index  $k$  of the mid-branch node in the internal index system; then it subtracts `jMin`, which transforms it in the corresponding external index; and finally takes the branch into account by adding `branch-1` (that is,  $-1$ ,  $0$  or  $1$  for left, mid and right branch). The `probability` method is easy enough: it selects the correct vector based on the branch and returns the probability for the  $i$ -th node. Since the vector indexes are zero-based, no conversion is needed.

Finally, the `underlying` method is implemented by `TrinomialTree` directly, since the branching doesn't store the relevant process information. The `Branching` class only needs to provide an inspector `jMin`, which the tree uses to determine the offset of the leftmost node from the center; it also provides a `jMax` inspector, as well as an `add` method which is used to build the branching. Such method should probably be called `push_back` instead; it takes the data for a single node (that is, mid-branch index and probabilities), adds them to the back of the corresponding vectors, and updates the information on the minimum and maximum indexes.

What remains now is for me to show (with the help of the listing that follows and, again, of [the previous figure](#)) how a `TrinomialTree` instance is built.

---

<sup>8</sup>There's no need to store the indexes of the left-branch and right-branch nodes, as they are always the neighbors of the mid-branch one.

---

**Construction of a TrinomialTree instance.**


---

```

TrinomialTree::TrinomialTree(
    const shared_ptr<StochasticProcess1D>& process,
    const TimeGrid& timeGrid,
    bool isPositive)
: Tree<TrinomialTree>(timeGrid.size()), dx_(1, 0.0),
  timeGrid_(timeGrid) {
    x0_ = process->x0();
    Size nTimeSteps = timeGrid.size() - 1;
    Integer jMin = 0, jMax = 0;

    for (Size i=0; i<nTimeSteps; i++) {
        Time t = timeGrid[i];
        Time dt = timeGrid.dt(i);

        Real v2 = process->variance(t, 0.0, dt);
        Volatility v = std::sqrt(v2);
        dx_.push_back(v*std::sqrt(3.0));

        Branching branching;
        for (Integer j=jMin; j<=jMax; j++) {
            Real x = x0_ + j*dx_[i];
            Real f = process->expectation(t, x, dt);
            Integer k = std::floor((f-x0_)/dx_[i+1] + 0.5);

            if (isPositive)
                while (x0_+(k-1)*dx_[i+1]<=0)
                    k++;

            Real e = f - (x0_ + k*dx_[i+1]);
            Real e2 = e*e, e3 = e*std::sqrt(3.0);

            Real p1 = (1.0 + e2/v2 - e3/v)/6.0;
            Real p2 = (2.0 - e2/v2)/3.0;
            Real p3 = (1.0 + e2/v2 + e3/v)/6.0;

            branching.add(k, p1, p2, p3);
        }
        branchings_.push_back(branching);
        jMin = branching.jMin();
        jMax = branching.jMax();
    }
}

```

---

As you saw a couple of pages back, the constructor takes the stochastic process for the underlying variable, a time grid, and a boolean flag. The number of times in the grid corresponds to the number of levels in the tree, so it is passed to the base `Tree` constructor; also, the time grid is stored and a vector `dx_`, which will store the distances between nodes at each level, is initialized. The first level has just one node, so there's no corresponding distance to speak of; thus, the first element of the vector is just set to 0. Other preparations include storing the initial value `x0_` of the underlying and the number of steps; and finally, declaring two variables `jMin` and `jMax` to hold the minimum and maximum node index. At the initial level, they both equal 0.

After this introduction, the tree is built recursively from one level to the next. For each step, we take the initial time `t` and the time step `dt` from the time grid. They're used as input to retrieve the variance of the process over the step, which is assumed to be independent of the value of the underlying (as for binomial trees, we have no way to enforce this). Based on the variance, we calculate the distance to be used at the next level and store it in the `dx_` vector;<sup>9</sup> and after this, we finally build a `Branching` instance.

To visualize the process, let's refer again to the figure. The code cycles over the nodes on the lower level, whose indexes range between the current values of `jMin` and `jMax`: in our case, that's -2 for  $A_1$  and 2 for  $A_5$ . For each node, we can calculate the underlying value  $x$  from the initial value `x0_`, the index  $j$ , and the distance `dx_[i]` between the nodes. From  $x$ , the process can give us the forward value  $f$  of the variable after `dt`; and having just calculated the distance `dx_[i+1]` between nodes on the upper level, we can find the index  $k$  of the node closest to  $f$ . As usual,  $k$  is an internal index; for the node  $A_4$  in the figure, whose forward value is  $F_4$ , the index  $k$  would be 0 corresponding to the center node  $B_3$ .

If the boolean flag `isPositive` is true, we have to make sure that no node corresponds to a negative or null value of the underlying; therefore, we check the value at the left target node (that would be the one with index  $k-1$ , since  $k$  is the index of the middle one) and if it's not positive, we increase  $k$  and repeat until the desired condition holds. In the figure, if the underlying were negative at node  $B_1$  then node  $A_1$  would branch to  $B_2$ ,  $B_3$  and  $B_4$  instead.

Finally, we calculate the three transition probabilities  $p_1$ ,  $p_2$  and  $p_3$  (the formulas are derived in [Brigo and Mercurio, 2006](#)) and store them in the current `Branching` instance together with  $k$ . When all the nodes are processed, we store the branching and update the values of `jMin` and `jMax` so that they range over the nodes on the upper level; the new values will be used for the next step of the main `for` loop (the one over time steps) in which the current upper level will become the lower level.

### 7.2.3 The `TreeLattice` class template

After this rather long detour on trees, we're now back to lattices. The `TreeLattice` class, shown in the next listing, inherits from `Lattice` and is used as a base class for lattices that are implemented

---

<sup>9</sup>The value of  $\sqrt{3}$  times the standard deviation is suggested by Hull and White for stability.

in terms of one or more trees. Of course, it should be called `TreeBasedLattice` instead; the shorter name was probably chosen before we discovered the marvels of automatic completion—or English grammar.

Sketch of the `TreeLattice` class template.

---

```

template <class Impl>
class TreeLattice : public Lattice,
    public CuriouslyRecurringTemplate<Impl> {
public:
    TreeLattice(const TimeGrid& timeGrid, Size n);
    void initialize(DiscretizedAsset& asset, Time t) const {
        Size i = t_.index(t);
        asset.time() = t;
        asset.reset(this->impl().size(i));
    }
    void rollback(DiscretizedAsset& asset, Time to) const {
        partialRollback(asset,to);
        asset.adjustValues();
    }
    void partialRollback(DiscretizedAsset& asset, Time to) const {
        Integer iFrom = Integer(t_.index(asset.time()));
        Integer iTo = Integer(t_.index(to));
        for (Integer i=iFrom-1; i>=iTo; --i) {
            Array newValues(this->impl().size(i));
            this->impl().stepback(i, asset.values(), newValues);
            asset.time() = t_[i];
            asset.values() = newValues;
            if (i != iTo) // skip the very last adjustment
                asset.adjustValues();
        }
    }
    void stepback(Size i, const Array& values,
                  Array& newValues) const {
        for (Size j=0; j<this->impl().size(i); j++) {
            Real value = 0.0;
            for (Size l=0; l<n_; l++) {
                value += this->impl().probability(i,j,l) *
                    values[this->impl().descendant(i,j,l)];
            }
            value *= this->impl().discount(i,j);
            newValues[j] = value;
        }
    }
}

```

```

Real presentValue(DiscretizedAsset& asset) const {
    Size i = t_.index(asset.time());
    return DotProduct(asset.values(), statePrices(i));
}
const Array& statePrices(Size i) const;
};
```

---

This class template acts as an adapter between the `Lattice` class, from which it inherits the interface, and the `Tree` class template which will be used for the implementation. Once again, we used CRTP (which wasn't actually needed for trees, but it is in this case); the behavior of the lattice is written in terms of a number of methods that must be defined in derived classes. For greater generality, there is no mention of trees in the `TreeLattice` class. It's up to derived classes to choose what kind of trees they should contain and how to use them.

The `TreeLattice` constructor is straightforward: it takes and stores the time grid and an integer `n` specifying the order of the tree (2 for binomial, 3 for trinomial and so on). It also performs a check or two, and initializes a couple of data members used for caching data; but I'll gloss over that here.

The interesting part is the implementation of the `Lattice` interface, which follows the outline I gave in a previous section. The `initialize` method calculates the index of the passed time on the stored grid, sets the asset time, and finally passes the number of nodes on the corresponding tree level to the asset's `reset` method. The number of nodes is obtained by calling the `size` method through CRTP; this is one of the methods that derived classes will have to implement, and (like all other such methods) has the same signature as the corresponding method in the tree classes.

The `rollback` and `partialRollback` methods perform the same work, with the only difference that `rollback` performs the adjustment at the final time and `partialRollback` doesn't. Therefore, it's only to be expected that the one is implemented in terms of the other; `rollback` performs a call to `partialRollback`, followed by another to the asset's `adjustValues` method.

The rollback procedure is spelled out in `partialRollback`: it finds the indexes of the current time and of the target time on the grid, and it loops from one to the other. At each step, it calls the `stepback` method, which implements the actual numerical work of calculating the asset values on the `i`-th level of the tree from those on level `i+1`; then it updates the asset and, at all steps except the last, calls its `adjustValues` method.

The implementation of the `stepback` method defines, by using it,<sup>10</sup> the interface that derived classes must implement. It determines the value of the asset at each node by combining the values at each descendant node, weighed by the corresponding transition probability; the result is further adjusted by discounting it. All in all, the required interface includes the `size` method, which I've already shown; the `probability` and `descendant` methods, with the same signature as the tree methods of the same name; and the `discount` method, which takes the indexes of the desired level and node and returns the discount factor between that node and its descendants (assumed to be independent of the particular descendant).

<sup>10</sup>That's currently the only sane way for us, since concepts are only available since C++20.

Finally, the `presentValue` method is implemented by returning the dot-product of the asset values by the state prices at the current time on the grid. I'll cheerfully ignore the way the state prices are calculated; suffice it to say that using them is somewhat more efficient than rolling the asset all the way back to  $t = 0$ .

Now, why does the `TreeLattice` implementation calls methods with the same signature as those of a tree (thus forcing derived classes to define them) instead of just storing a tree and calling its methods directly? Well, that's the straightforward thing to do if you have just an underlying tree; and in fact, most one-dimensional lattices will just forward the calls to the tree they store. However, it wouldn't work for other lattices (say, two-dimensional ones); and in that case, the wrapper methods used in the implementation of `stepback` allow us to adapt the underlying structure, whatever that is, to their common interface.

The library contains instances of both kinds of lattices. Most—if not all—of those of the straightforward kind inherit from the `TreeLattice1D` class template, shown in the next listing. It doesn't define any of the methods required by `TreeLattice`; and the method it does implement (the `grid` method, defined as pure virtual in the `Lattice` base class) actually requires another one, namely, the `underlying` method. All in all, this class does little besides providing a useful categorization; the storage and management of the underlying tree is, again, left to derived classes.<sup>11</sup>

Interface of the `TreeLattice1D` class template.

---

```
template <class Impl>
class TreeLattice1D : public TreeLattice<Impl> {
public:
    TreeLattice1D(const TimeGrid& timeGrid, Size n);
    Disposable<Array> grid(Time t) const;
};
```

---

One such class is the inner `OneFactorModel::ShortRateTree` class, shown in the listing that follows. Its constructor takes a trinomial tree, built by any specific short-rate model according to its dynamics; an instance of the `ShortRateDynamics` class, which I'll gloss over; and a time grid, which could have been extracted from the tree so I can't figure out why we pass it instead. The grid is passed to the base-class constructor, together with the order of the tree (which is 3, of course); the tree and the dynamics are stored as data members.

---

<sup>11</sup>One might argue for including a default implementation of the required methods in `TreeLattice1D`. This would probably make sense; it would make it easier to implement derived classes in the most common cases, and could be overridden if a specific lattice needed it.

Implementation of the `OneFactorModel::ShortRateTree` class.

---

```

class OneFactorModel::ShortRateTree
    : public TreeLattice1D<OneFactorModel::ShortRateTree> {
public:
    ShortRateTree(
        const shared_ptr<TrinomialTree>& tree,
        const shared_ptr<ShortRateDynamics>& dynamics,
        const TimeGrid& timeGrid)
    : TreeLattice1D<OneFactorModel::ShortRateTree>(timeGrid, 3),
      tree_(tree), dynamics_(dynamics) {}
    Size size(Size i) const {
        return tree_->size(i);
    }
    Real underlying(Size i, Size index) const {
        return tree_->underlying(i, index);
    }
    Size descendant(Size i, Size index, Size branch) const {
        return tree_->descendant(i, index, branch);
    }
    Real probability(Size i, Size index, Size branch) const {
        return tree_->probability(i, index, branch);
    }
    DiscountFactor discount(Size i, Size index) const {
        Real x = tree_->underlying(i, index);
        Rate r = dynamics_->shortRate(timeGrid()[i], x);
        return std::exp(-r*timeGrid().dt(i));
    }
private:
    shared_ptr<TrinomialTree> tree_;
    shared_ptr<ShortRateDynamics> dynamics_;
};
```

---

As is to be expected, most of the required interface is implemented by forwarding the call to the corresponding tree method. The only exception is the `discount` method, which doesn't have a corresponding tree method; it is implemented by asking the tree for the value of its underlying value at the relevant node, by retrieving the short rate from the dynamics, and by calculating the corresponding discount factor between the time of the node and the next time on the grid.

Note that, by modifying the dynamics, it is possible to change the value of the short rate at each node while maintaining the structure of the tree unchanged. This is done in a few models in order to fit the tree to the current interest-rate term structure; the `ShortRateTree` class provides another

constructor, not shown here, that takes additional parameters to perform the fitting procedure.

\* \* \*

As an example of the second kind of lattice, have a look at the `TreeLattice2D` class template, shown in the next listing. It acts as base class for lattices with two underlying variables, and implements most of the methods required by `TreeLattice`.<sup>12</sup>

Implementation of the `TreeLattice2D` class template.

---

```
template <class Impl, class T = TrinomialTree>
class TreeLattice2D : public TreeLattice<Impl> {
public:
    TreeLattice2D(const shared_ptr<T>& tree1,
                  const shared_ptr<T>& tree2,
                  Real correlation)
        : TreeLattice<Impl>(tree1->timeGrid(),
                              T::branches*T::branches),
          tree1_(tree1), tree2_(tree2), m_(T::branches,T::branches),
          rho_(std::fabs(correlation)) { ... }
    Size size(Size i) const {
        return tree1_->size(i)*tree2_->size(i);
    }
    Size descendant(Size i, Size index, Size branch) const {
        Size modulo = tree1_->size(i);

        Size index1 = index % modulo;
        Size index2 = index / modulo;
        Size branch1 = branch % T::branches;
        Size branch2 = branch / T::branches;

        modulo = tree1_->size(i+1);
        return tree1_->descendant(i, index1, branch1) +
               tree2_->descendant(i, index2, branch2)*modulo;
    }
    Real probability(Size i, Size index, Size branch) const {
        Size modulo = tree1_->size(i);

        Size index1 = index % modulo;
        Size index2 = index / modulo;
```

<sup>12</sup>In this, it differs from `TreeLattice1D` which didn't implement any of them. We might have had a more symmetric hierarchy by leaving `TreeLattice2D` mostly empty and moving the implementation to a derived class. At this time, though, it would sound a bit like art for art's sake.

```

    Size branch1 = branch % T::branches;
    Size branch2 = branch / T::branches;

    Real prob1 = tree1_->probability(i, index1, branch1);
    Real prob2 = tree2_->probability(i, index2, branch2);
    return prob1*prob2 + rho_*(m_[branch1][branch2])/36.0;
}

protected:
    shared_ptr<T> tree1_, tree2_;
    Matrix m_;
    Real rho_;
};


```

---

The two variables are modeled by correlating the respective trees. Now, I'm sure that any figure I might draw would only add to the confusion. However, the idea is that the state of the two variables is expressed by a pair of node from the respective trees; that the transitions to be considered are those from pair to pair; and that all the possibilities are enumerated so that they can be retrieved by means a single index and thus can match the required interface.

For instance, let's take the case of two trinomial trees. Let's say we're at level  $i$  (the two trees must have the same time grid, or all bets are off). The first variable has a value that corresponds to node  $j$  on its tree, while the second sits on node  $k$ . The structure of the first tree tells us that on next level, the first variable might go to nodes  $j'_0$ ,  $j'_1$  or  $j'_2$  with different probabilities; the second tree gives us  $k'_0$ ,  $k'_1$  and  $k'_2$  as target nodes for the second variable. Seen as transition between pairs, this means that we're at  $(j, k)$  on the current level and that on the next level we might go to any of  $(j'_0, k'_0)$ ,  $(j'_1, k'_0)$ ,  $(j'_2, k'_0)$ ,  $(j'_0, k'_1)$ ,  $(j'_1, k'_1)$ , and so on until  $(j'_2, k'_2)$  for a grand total of  $3 \times 3 = 9$  possibilities. By enumerating the pairs in lexicographic order like I just did, we can give  $(j'_0, k'_0)$  the index 0,  $(j'_1, k'_0)$  the index 1, and so on until we give the index 8 to  $(j'_2, k'_2)$ . In the same way, if on a given level there are  $n$  nodes on the first tree and  $m$  on the second, we get  $n \times m$  pairs that, again, can be enumerated in lexicographic order: the pair  $(j, k)$  is given the index  $k \times n + j$ .

At this point, the implementation starts making sense. The constructor of `TreeLattice2D` takes and stores the two underlying trees and the correlation between the two variables; the base-class `TreeLattice` constructor is passed the time grid, taken from the first tree, and the order of the lattice, which equals the product of the orders of the two trees; for two trinomial trees, this is  $3 \times 3 = 9$  as above.<sup>13</sup> The constructor also initializes a matrix `m_` that will be used later on.

The size of the lattice at a given level is the product  $n \times m$  of the sizes of the two trees, which translates in a straightforward way into the implementation of the `size` method.

Things get more interesting with the two following methods. The `descendant` method takes the level  $i$  of the tree; the index of the lattice node, which is actually the index of a pair  $(j, k)$  among all those available; and the index of the branch to take, which by the same token is a pair of branches.

<sup>13</sup>The current implementation assumes that the two trees are of the same type, but it could easily be made to work with two trees of different orders.

The first thing it does is to extract the actual pairs. As I mentioned, the passed index equals  $k \times n + j$ , which means that the two underlying indexes can be retrieved as `index%n` and `index/n`. The same holds for the two branches, with  $n$  being replaced by the order of the first tree. Having all the needed indexes and branches, the code calls the `descendant` method on the two trees, obtaining the indexes  $j'$  and  $k'$  of the descendant nodes; then it retrieves the size  $n'$  of the first tree at the next level; and finally returns the combined index  $k' \times n' + j'$ .

Up to a certain point, the `probability` method performs the same calculations; that is, until it retrieves the two probabilities from the two underlying trees. If the two variables were not correlated, the probability for the transition would then be the product of the two probabilities. Since this is not the case, a correction term is added which depends from the passed correlation (of course) and also from the chosen branches. I'll gloss on the value of the correction as I already did on several other formulas.

This completes the implementation. Even though it contains a lot more code than its one-dimensional counterpart, `TreeLattice2D` is still an incomplete class. Actual lattices will have to inherit from it, close the CRTP loop, and implement the missing `discount` method.

\* \* \*

I'll close this section by mentioning one feature that is currently missing from lattices, but would be nice to have. If you turn back to the listing of the `TreeLattice` class, you'll notice that the `stepback` method assumes that the values we're rolling back are cash values; that is, it always discounts them. It could also be useful to roll values back without discounting. For instance, the probability that an option be exercised could be calculated by rolling it back on the trees without discounting, while adjusting it to 1 on the nodes where the exercise condition holds.

## 7.3 Tree-based engines

As you might have guessed, a tree-based pricing engine will perform few actual computations; its main job will rather be to instantiate and drive the needed discretized asset and lattice.<sup>14</sup>

### 7.3.1 Example: callable fixed-rate bonds

As an example, I'll sketch the implementation of a tree-based pricing engine for callable fixed-rate bonds. For the sake of brevity, I'll skip the description of the `CallableBond` class.<sup>15</sup> Instead, I'll just show its inner `arguments` and `results` classes, which act as its interface with the pricing engine and which you can see in the listing below together with the corresponding `engine` class. If you're interested in a complete implementation, you can look for it in QuantLib's experimental folder.

<sup>14</sup>If you're pattern-minded, you can have your pick here. This implementation has suggestions of the Adapter, Mediator, or Facade pattern, even though it doesn't match any of them exactly.

<sup>15</sup>To be specific, the class name should be `CallableFixedRateBond`; but that would get old very quickly here, so please allow me to use the shorter name.

Interface of the `CallableBond` inner classes.

---

```

class CallableBond::arguments : public PricingEngine::arguments {
    public:
        std::vector<Date> couponDates;
        std::vector<Real> couponAmounts;
        Date redemptionDate;
        Real redemptionAmount;
        std::vector<Callability::Type> callabilityTypes;
        std::vector<Date> callabilityDates;
        std::vector<Real> callabilityPrices;
        void validate() const;
};

class CallableBond::results : public Instrument::results {
    public:
        Real settlementValue;
};

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

```

---

Now, let's move into engine territory. In order to implement the behavior of the instrument, we'll need a discretized asset; namely, the `DiscretizedCallableBond` class, shown in the next listing.

Implementation of the `DiscretizedCallableBond` class.

---

```

class DiscretizedCallableBond : public DiscretizedAsset {
    public:
        DiscretizedCallableBond(const CallableBond::arguments& args,
                               const Date& referenceDate,
                               const DayCounter& dayCounter)
            : arguments_(args) {
                redemptionTime_ =
                    dayCounter.yearFraction(referenceDate,
                                             args.redemptionDate);

                couponTimes_.resize(args.couponDates.size());
                for (Size i=0; i<couponTimes_.size(); ++i)
                    couponTimes_[i] =
                        dayCounter.yearFraction(referenceDate,
                                             args.couponDates[i]);
            // same for callability times

```

---

```
    }

    std::vector<Time> mandatoryTimes() const {
        std::vector<Time> times;

        Time t = redemptionTime_;
        if (t >= 0.0)
            times.push_back(t);
        // also add non-negative coupon times and callability times

        return times;
    }

    void reset(Size size) {
        values_ = Array(size, arguments_.redemptionAmount);
        adjustValues();
    }

protected:
    void preAdjustValuesImpl();
    void postAdjustValuesImpl();
private:
    CallableBond::arguments arguments_;
    Time redemptionTime_;
    std::vector<Time> couponTimes_;
    std::vector<Time> callabilityTimes_;
    void applyCallability(Size i);
    void addCoupon(Size i);
};

void DiscretizedCallableBond::preAdjustValuesImpl() {
    for (Size i=0; i<callabilityTimes_.size(); i++) {
        Time t = callabilityTimes_[i];
        if (t >= 0.0 && isOnTime(t)) {
            applyCallability(i);
        }
    }
}

void DiscretizedCallableBond::postAdjustValuesImpl() {
    for (Size i=0; i<couponTimes_.size(); i++) {
        Time t = couponTimes_[i];
        if (t >= 0.0 && isOnTime(t)) {
            addCoupon(i);
        }
    }
}
```

```

}

void DiscretizedCallableBond::applyCallability(Size i) {
    switch (arguments_.callabilityTypes[i]) {
        case Callability::Call:
            for (Size j=0; j<values_.size(); j++) {
                values_[j] =
                    std::min(arguments_.callabilityPrices[i],
                             values_[j]);
            }
            break;
        case Callability::Put:
            for (Size j=0; j<values_.size(); j++) {
                values_[j] =
                    std::max(arguments_.callabilityPrices[i],
                             values_[j]);
            }
            break;
        default:
            QL_FAIL("unknown callability type");
    }
}

void DiscretizedCallableBond::addCoupon(Size i) {
    values_ += arguments_.couponAmounts[i];
}

```

---

To prevent much aggravation, its constructor takes and stores an instance of the `arguments` class. This avoids having to spell out the list of needed data in at least three places: the declaration of the data members, the constructor, and the client code that instantiates the discretized asset. Besides the `arguments` instance, the constructor is also passed a reference date and a day counter that are used in its body to convert the several bond dates into corresponding times. (The conversion is somewhat verbose, which suggests that we might be missing an abstraction here. However, “time converter” sounds a bit too vague; maybe “time scale”? Well, if you find it, please let me know. The thing has been bugging me for a while.)

Next comes the required `DiscretizedAsset` interface. The `mandatoryTimes` method collects the redemption time, the coupon times, and the callability times filtering out the negative ones; and the `reset` method resizes the array of the values, sets each one to the redemption amount, and proceeds to perform the needed adjustments—that is, the more interesting part of the class.

Being rather specialized, it is pretty unlikely that this class will be composed with others; therefore, it doesn’t really matter in this case whether the adjustments go into `preAdjustValuesImpl` or

`postAdjustValuesImpl`. However, for sake of illustration, I'll separate the callability from the coupon payments and manage them as pre- and post-adjustment, respectively.

The `preAdjustValuesImpl` loops over the callability times, checks whether any of them equals the current time, and calls the `applyCallability` method if this is the case. The `postAdjustValuesImpl` does the same, but checking the coupon times and calling the `addCoupon` method instead.

The `applyCallability` method is passed the index of the callability being exercised; it checks its type (both callable and puttable bonds are supported) and sets the value at each node to the value after exercise. The logic is as you expect: at each node, given the estimated value of the rest of the bond (that is, the current asset value) and the exercise premium, the issuer will choose the lesser of the two values while the holder will choose the greater. The `addCoupon` method is simpler, and just adds the coupon amount to each of the values.

As you might have noticed, this class assumes that the exercise dates coincide with the coupon dates; it won't work if an exercise date is a few days before a coupon payment (the coupon amount would be added to the asset values before the exercise condition is checked). Of course, this is often the case, and it should be accounted for. Currently, the library implementation sidesteps the problem by adjusting each exercise date so that it equals the nearest coupon date. A better choice would be to detect which coupons are affected; each of them would be put into a new asset, rolled back until the relevant exercise time, and added after the callability adjustment.

Finally, the listing below shows the `TreeCallableBondEngine` class. Its constructor takes and stores a handle to a short-rate model that will provide the lattice, the total number of time steps we want the lattice to have, and an optional reference date and day counter; the body just registers to the handle.

Sketch of the `TreeCallableBondEngine` class.

---

```

class TreeCallableBondEngine : public CallableBond::engine {
    public:
        TreeCallableBondEngine(
            const Handle<ShortRateModel>& model,
            const Size timeSteps,
            const Date& referenceDate = Date(),
            const DayCounter& dayCounter = DayCounter());
        : model_(model), timeSteps_(timeSteps),
          referenceDate_(referenceDate), dayCounter_(dayCounter) {
            registerWith(model_);
        }
    void calculate() const {
        Date referenceDate;
        DayCounter dayCounter;

        // try to extract the reference date and the day counter
        // from the model, use the stored ones otherwise.
    }
}
```

```

DiscretizedCallableBond bond(arguments_,
                           referenceDate,
                           dayCounter);

std::vector<Time> times = bond.mandatoryTimes();
TimeGrid grid(times.begin(), times.end(), timeSteps_);
shared_ptr<Lattice> lattice = model_->tree(grid);

Time redemptionTime =
    dayCounter.yearFraction(referenceDate,
                            arguments_.redemptionDate);
bond.initialize(lattice, redemptionTime);
bond.rollback(0.0);
results_.value = bond.presentValue();
}
};


```

---

The `calculate` method is where everything happens. By the time it is called, the engine arguments have been filled by the instrument, so that base is covered; the other data we need are a date and a day counter for time conversion. Not all short-rate models can provide them, so, in a boring few lines of code not shown here, the engine tries to downcast the model to some specific class that does; if it fails, it falls back to using the ones optionally passed to the constructor.

At that point, the actual calculations can begin. The engine instantiates the discretized bond, asks it for its mandatory times, and uses them to build a time grid; then, the grid is passed to the model which returns a corresponding lattice based on the short-rate dynamics. All that remains is to initialize the bond at its redemption time (which in the current code is recalculated explicitly, but could be retrieved as the largest of the mandatory times), roll it back to the present time, and read its value.

# 8. The finite-difference framework

The last framework I'll be writing about was, in fact, the first to appear in the library. Of course, it was not a framework back then; it was just a bunch of code inside a single option pricer.

Since then, the reusable parts of the code were reorganized and grew in complexity—probably to the point where it's hard to find one's way inside the architecture. A few years ago, a new version of the framework was contributed adding 2-dimensional finite-difference models (which were absent in the first version) and sporting a more modular architecture.

The two versions of the framework still coexist in the library. In this chapter, I'll describe both of them—while hoping for the new one to replace the first some day.

## 8.1 The old framework

As I mentioned in [chapter 7](#), our master plan to create a common framework for trees and finite-difference models never came to fruition. Instead, the finite-difference code stayed closer to its underlying mathematics and implemented concepts such as differential operators, boundary conditions and so on. I'll tackle one of them in each of the following subsections.

### 8.1.1 Differential operators

If I were to start at the very beginning, I'd describe the `Array` class here; but they behave as you can imagine, so you'll forgive me if I leave its description to [appendix A](#) and just note that it's used here to discretize the function  $f(x)$  on a grid  $\{x_0 \dots x_{N-1}\}$  as the array  $\mathbf{f}$ , with  $\mathbf{f}_i = f(x_i)$  for any  $i$ .

Onwards to operators. Now, I'm probably preaching to the choir, so I'll keep the math to a minimum to avoid telling you lots of stuff you know already.<sup>1</sup> In short: a differential operator transforms a function  $f(x)$  in one of its derivatives, say,  $f'(x)$ . The discretized version transforms  $\mathbf{f}$  into  $\mathbf{f}'$ , and since differentiation is linear, it can be written as a matrix. In general, the operator doesn't give the exact discretization of the derivative but just an approximation; that is,  $\mathbf{f}'_i = f'(x_i) + \epsilon_i$  where the error terms can be reduced by decreasing the spacing of the grid.

QuantLib provides differential operators for the first derivative  $\partial/\partial x$  and the second derivative  $\partial^2/\partial x^2$ . They are called  $D_0$  and  $D_+D_-$ , respectively, and are defined as:

$$\frac{\partial f}{\partial x}(x_i) \approx D_0 \mathbf{f}_i = \frac{\mathbf{f}_{i+1} - \mathbf{f}_{i-1}}{2h} \quad \frac{\partial^2 f}{\partial x^2}(x_i) \approx D_+D_- \mathbf{f}_i = \frac{\mathbf{f}_{i+1} - 2\mathbf{f}_i + \mathbf{f}_{i-1}}{h^2}$$

---

<sup>1</sup>And if you don't, there's no way I can tell you all you need. You can refer to any of the several books on finite differences; for instance, ([Duffy, 2006](#)).

where  $h = x_i - x_{i-1}$  (we're assuming that the grid is evenly spaced). Taylor expansion shows that the error is  $o(h^2)$  for both of them.

As you can see, the value of the derivative at any given index  $i$  (except the first and last, that I'll ignore now) only depends from the values of the function at the same index, the one that precedes it, and the one that follows. This makes  $D_0$  and friends tridiagonal: the structure of such operators can be sketched as

$$\begin{pmatrix} d_0 & u_0 & & & \\ l_0 & d_1 & u_1 & & \\ & l_1 & d_2 & u_2 & \\ & & \ddots & \ddots & \ddots \\ & & & l_{n-4} & d_{n-3} & u_{n-3} \\ & & & & l_{n-3} & d_{n-2} & u_{n-2} \\ & & & & & l_{n-2} & d_{n-1} \end{pmatrix}$$

They have a number of desirable properties, including the fact that a linear combination of tridiagonal operators (such as the one found, say, in the Black-Scholes-Merton formula) is still tridiagonal; thus, instead of using a generic `Matrix` class,<sup>2</sup> we made them instances of a specific class called `TridiagonalOperator` and shown in the following listing.

Partial interface of the `TridiagonalOperator` class.

---

```
class TridiagonalOperator {
    Size n_;
    Array diagonal_, lowerDiagonal_, upperDiagonal_;

public:
    typedef Array array_type;

    explicit TridiagonalOperator(Size size = 0);
    TridiagonalOperator(const Array& low,
                        const Array& mid,
                        const Array& high);

    static Disposable<TridiagonalOperator>
    identity(Size size);

    Size size() const;
    const Array& lowerDiagonal() const;
    const Array& diagonal() const;
    const Array& upperDiagonal() const;
```

---

<sup>2</sup>The `Matrix` class does exist, but is not used here.

```

void setFirstRow(Real, Real);
void setMidRow(Size, Real, Real, Real);
void setMidRows(Real, Real, Real);
void setLastRow(Real, Real);

Disposable<Array> applyTo(const Array& v) const;
Disposable<Array> solveFor(const Array& rhs) const;
void solveFor(const Array& rhs, Array& result) const;

private:
    friend Disposable<TridiagonalOperator>
    operator+(const TridiagonalOperator&,
               const TridiagonalOperator&);
    friend Disposable<TridiagonalOperator>
    operator*(Real, const TridiagonalOperator&);
    // other operators
};
```

---

As you can see from the private members, the representation of the operator reflects its tridiagonal structure: instead of storing all the mostly null elements of the matrix, we just store three arrays corresponding to the diagonal, lower diagonal, and upper diagonal; that is, the  $d_i$ ,  $l_i$  and  $u_i$  of the earlier sketch.

The constructors and the first bunch of methods deal with building and inspecting the operator. The first constructor sets the size of the matrix (that is, of the three underlying arrays) and trusts the values to be filled later, by means of the provided setters: referring to the figure again, the `setFirstRow` method sets  $d_0$  and  $u_0$ , the `setLastRow` method sets  $l_{n-2}$  and  $d_{n-1}$ , `setMidRow` sets  $l_{i-1}$ ,  $d_i$  and  $u_i$  for a given  $i$ , and the convenience method `setMidRows` sets them for all  $i$ . The second constructor sets all three arrays in one fell swoop; and a factory method, `identity`, works as a third constructor by building and returning an operator with 1 on the diagonal and 0 elsewhere.<sup>3</sup> The inspectors do the obvious thing and return each the corresponding data member.

The `applyTo` and `solveFor` methods are the meat of the interface. Given a tridiagonal operator  $L$  and an array  $u$ ,  $L.\text{applyTo}(u)$  returns the result  $Lu$  of applying  $L$  to  $u$ , while  $L.\text{solveFor}(u)$  returns the array  $v$  such that  $u = Lv$ .<sup>4</sup> Both operations will be used later, and for a tridiagonal operator they both have the nice property to be  $O(N)$  in time, that is, to only take time proportional to the size  $N$  of the operator. The algorithms were taken from the classic *Numerical Recipes in C* (Press *et al*, 1992). Not the code, of course: besides not being free, the original code had a number of endearing old conventions (such as numbering arrays from 1, like in Downton Abbey) that wouldn't have matched the rest of the library.

<sup>3</sup>As I write, the returned operator is wrapped in a `Disposable` class template, in a possibly misguided attempt to avoid a few copies: see [appendix A](#) for details. This might change in the future.

<sup>4</sup>The overloaded call  $L.\text{solveFor}(u, v)$  does the same while avoiding an allocation.

Finally, the class defines a few operators. They make it possible to write numerical expressions such as  $2*L1 + L2$  that might be used to compose tridiagonal operators. To this effect, the library also provides a few building blocks such as the  $D_0$  and  $D_+D_-$  operators I mentioned earlier; for instance, given the Black-Scholes equation, written in operator form as

$$\frac{\partial f}{\partial t} = \left[ -\left( r - q - \frac{\sigma^2}{2} \right) \frac{\partial}{\partial x} - \frac{\sigma^2}{2} \frac{\partial^2}{\partial x^2} + rI \right] f \equiv L_{BS}f$$

one could define the corresponding  $L_{BS}$  operator as

```
TridiagonalOperator L_BS = -(r-q-sigma*sigma/2)*DZero(N)
                           -(sigma*sigma/2)*DPlusDMinus(N)
                           -r*TridiagonalOperator::identity(N);
```

In this case, though, we didn't eat our own dog food. The library does define a Black-Scholes operator, but not through composition as above.

A final note: while they're probably the most performant, tridiagonal operators are not the only ones that could be used in a finite-difference scheme. However, there's no base class for such operators. The interface that they should implement (that is, `applyTo` and `solveFor`) is only defined implicitly, through its use in the template classes that we'll see in the next subsection. In hindsight, defining such a base class wouldn't have degraded performance; the cost of a virtual call to `applyTo` or `solveFor` would have been negligible, compared to the cost of the actual calculation. But we were eagerly reading about template numerical techniques at that time; and like the two guys in the Wondermark strip, we decided to jingle all the way.<sup>5</sup>

### 8.1.2 Evolution schemes

In a partial differential equation  $\partial f / \partial t = Lf$ , the operators in the previous subsection provide a discretization of the derivatives on the right-hand side. Evolution schemes discretize the time derivative on the left-hand side instead.

As you know, finite-difference methods in finance start from a known state  $\mathbf{f}(T)$  at maturity (the payoff of the derivative) and evolve backwards to today's date. At each step, we need to evaluate  $\mathbf{f}(t)$  based on  $\mathbf{f}(t + \Delta t)$ .

The simplest way is to approximate the equation above as

$$\frac{\mathbf{f}(t + \Delta t) - \mathbf{f}(t)}{\Delta t} = L \cdot \mathbf{f}(t + \Delta t)$$

which simplifies to  $\mathbf{f}(t) = \mathbf{f}(t + \Delta t) - \Delta t \cdot L \cdot \mathbf{f}(t + \Delta t)$ , or  $\mathbf{f}(t) = (I - \Delta t \cdot L) \cdot \mathbf{f}(t + \Delta t)$ . It's a simple matrix multiplication, and updating the function array translates into

---

<sup>5</sup>See <http://wondermark.com/779/>.

```
a = (I - dt*L).applyTo(a);
```

(in actual code, of course, the `I + dt*L` operator would be precalculated and stored across time steps).

A slightly less simple way is to write

$$\frac{\mathbf{f}(t + \Delta t) - \mathbf{f}(t)}{\Delta t} = L \cdot \mathbf{f}(t)$$

which results into  $(I + \Delta t \cdot L) \cdot \mathbf{f}(t) = \mathbf{f}(t + \Delta t)$ . This is a more complex step, since it requires solving a linear system (or inverting the operator, which is the same); but given the interface of our operators, the update translates equally simply into

```
a = (I + dt*L).solveFor(a);
```

Now, this is where using tridiagonal operators pays off. Let me oversimplify, so I can keep a long story short: schemes of the first kind (called *explicit* schemes) are simpler but less stable. *Implicit* schemes, instead (those of the second kind), are more stable and can be used with larger time steps. If the operator were a generic one, the price for an implicit step would be  $O(N^3)$  in the size of the grid, making it unpractical when compared to the  $O(N^2)$  cost for an explicit step. For tridiagonal operator, though, both methods cost  $O(N)$ , which allows us to reap the benefits of explicit steps with no additional price.

Going back to the library: the two schemes above (called *explicit Euler* and *implicit Euler* scheme, respectively) are both available. They are two particular specializations of a generic `MixedScheme` class template, shown in the listing that follows, which models the discretization scheme

$$\frac{\mathbf{f}(t + \Delta t) - \mathbf{f}(t)}{\Delta t} = L \cdot [(1 - \theta) \cdot \mathbf{f}(t + \Delta t) + \theta \cdot \mathbf{f}(t)]$$

where an implicit and explicit step are mixed. As you can see from the formula,  $\theta = 0$  gives the explicit Euler scheme and  $\theta = 1$  gives implicit Euler; any other value in the middle can be used (for instance,  $1/2$  gives the Crank-Nicolson scheme).

Sketch of the `MixedScheme` class template and a derived class.

---

```
template <class Operator>
class MixedScheme {
    public:
        typedef OperatorTraits<Operator> traits;
        typedef typename traits::operator_type operator_type;
        typedef typename traits::array_type array_type;
        typedef typename traits::bc_set bc_set;

    MixedScheme(const operator_type& L,
```

```

        Real theta,
        const bc_set& bcs)
: L_(L), I_(operator_type::identity(L.size())),
dt_(0.0), theta_(theta) , bcs_(bcs) {}

void setStep(Time dt) {
    dt_ = dt;
    if (theta_ != 1.0) // there is an explicit part
        explicitPart_ = I_-((1.0-theta_) * dt_)*L_;
    if (theta_ != 0.0) // there is an implicit part
        implicitPart_ = I_+(theta_ * dt_)*L_;
}
void step(array_type& a, Time t) {
    if (theta_ != 1.0) {
        a = explicitPart_.applyTo(a);
    if (theta_ != 0.0) {
        implicitPart_.solveFor(a, a);
    }
protected:
operator_type L_, I_, explicitPart_, implicitPart_;
Time dt_;
Real theta_;
bc_set bcs_;
};

template <class Operator>
class ExplicitEuler : public MixedScheme<Operator> {
public:
ExplicitEuler(const operator_type& L,
              const bc_set& bcs)
: MixedScheme<Operator>(L, 0.0, bcs) {}
};
```

---

As I mentioned before, schemes take the type of their differential operators as a template argument. The `MixedScheme` class does that as well, and extracts a number of types from the operator class by means of the `OperatorTraits` template (that I'll describe later on).

The constructor of the `MixedScheme` class takes the differential operator  $L$ , the value of  $\theta$ , and a set of boundary conditions (more on that later) and stores them. Another method, `setStep`, stores the  $\Delta t$  to be used as time step and precomputes the operators  $I - (1 - \theta) \cdot \Delta t \cdot L$  and  $I + \theta \cdot \Delta t \cdot L$  on which the `applyTo` and `solveFor` methods will be called; the two special cases  $\theta = 0$  and  $\theta = 1$  are checked in order to avoid unnecessary computations. This is done in a separate method in case we needed to change the time step in the middle of a simulation; by being able to reset the time step, we can

reuse the same scheme instance.

Finally, the `step` method performs the actual evolution of the function array. The implementation shown in the listing is just a basic sketch that I'll be improving in the next subsections; here, it just performs the explicit and implicit part of the scheme in this order (again, avoiding unnecessary operations).

The few schemes provided by the library (such as the `ExplicitEuler`, shown in the listing) are all specializations of the `MixedScheme` class.<sup>6</sup> Again, there's no base class for an evolution scheme; other parts of the framework will take them as a template argument and expect them to provide the `setStep` and `step` methods.

### 8.1.3 Boundary conditions

As I said, there's more going on in the `step` method. To begin with, we'll have to enforce boundary conditions at either end of the grid; for instance, we might want to ensure that an option has price close to 0 at the grid boundary which is deep out of the money and derivative close to 1 at the boundary which is deep in the money. Such conditions are modeled by the `BoundaryCondition` class, shown in the listing below.

Interface of the `BoundaryCondition` class template.

---

```
template <class Operator>
class BoundaryCondition {
public:
    typedef Operator operator_type;
    typedef typename Operator::array_type array_type;

    enum Side { None, Upper, Lower };

    virtual ~BoundaryCondition() {}

    virtual void applyBeforeApplying(operator_type&) const = 0;
    virtual void applyAfterApplying(array_type&) const = 0;
    virtual void applyBeforeSolving(operator_type&,
                                    array_type& rhs) const = 0;
    virtual void applyAfterSolving(array_type&) const = 0;
};
```

---

As you see, the class uses runtime polymorphism. This is because we wanted to store collections of them, possibly of different types (you might remember the `bc_set` `typedef` from the listing of the `MixedScheme` class), and most suitable containers are homogeneous and thus require a common

<sup>6</sup>I'm just talking about the old framework here; the new framework has different ones.

base class.<sup>7</sup> Well, that's not exactly true: even at that time, we could have used `std::pair` if we had decided to limit ourselves to the 1-D case. We didn't, though, and in hindsight I think it avoided adding more complexity to the framework.

The interface of the class is, well, peculiar. Glossing over the `Side` enumeration (which is meant to specify a grid side in 1-D models, and thus too specific), methods like `applyBeforeApplying` are plausible contenders for the worst name in the library. The idea here is that the boundary condition can be enforced, or “applied”, in different ways. One can wait for the operator's `applyTo` or `solveFor` to execute, and then modify the result; this would be implemented by `applyAfterApplying` and `applyAfterSolving`, respectively. Another possibility is to set up the operator and the input array beforehand, so that the result of the operation will satisfy the condition; this is done in `applyBeforeApplying` and `applyBeforeSolving`. (For some reason, the former only takes the operator. It might have been an oversight, due to the fact that the existing conditions didn't require to modify the array.)

As an example, look at the `DirichletBC` class in the listing below. It implements a simple Dirichlet boundary condition, i.e., one in which the function value at a given end of the grid must equal a given constant value.

Implementation of the `DirichletBC` class.

---

```
class DirichletBC
    : public BoundaryCondition<TridiagonalOperator> {
public:
    DirichletBC(Real value, Side side);

    void applyBeforeApplying(TridiagonalOperator& L) const {
        switch (side_) {
            case Lower:
                L.setFirstRow(1.0, 0.0);
                break;
            case Upper:
                L.setLastRow(0.0, 1.0);
                break;
        }
    }

    void applyAfterApplying(Array& u) const {
        switch (side_) {
            case Lower:
                u[0] = value_;
                break;
            case Upper:
                u[u.size()-1] = value_;
        }
    }
}
```

---

<sup>7</sup>`std::tuple` wasn't even a gleam in the standard committee's eye, and handling it requires template metaprogramming, which I'd try to avoid unless necessary—even in this time and age.

```

        break;
    }
}

void applyBeforeSolving(TridiagonalOperator& L,
                        Array& rhs) const {
    switch (side_) {
        case Lower:
            L.setFirstRow(1.0,0.0);
            rhs[0] = value_;
            break;
        case Upper:
            L.setLastRow(0.0,1.0);
            rhs[rhs.size()-1] = value_;
            break;
    }
}

void applyAfterSolving(Array&) const {}

};
```

---

As you can see, it is no longer a class template; when implemented, boundary conditions are specialized for a given operator—not surprisingly, because they have to know how to access and modify it. In this case, we’re using the tridiagonal operator I described in an earlier subsection.

The constructor is simple enough: it takes the constant value and the side of the grid to which it must be applied, and stores them.

As I said, the `applyBeforeApplying` method must set up the operator `L` so that the result of `L.apply(u)`, or  $\mathbf{u}' = L \cdot \mathbf{u}$ , satisfies the boundary condition. To do this, it sets the first (last) row to the corresponding row of the identity matrix; this ensures that the first (last) element of  $\mathbf{u}'$  equal the corresponding element of  $\mathbf{u}$ . Given that the array satisfied the boundary condition at the previous step, it keeps satisfying it at this step. Relying on this is a bit unsafe (for instance, it would break if the value changed at different times, or if any other event modified the array values) but it’s the best one can do without accessing the input array. The `applyAfterApplying` method is simpler and safer: it just sets the value of the output array directly.

The `applyBeforeSolving` method works as its sibling `applyBeforeApplying`, but it can also access the input array, so it can ensure that it contains the correct value. Finally, and surprisingly enough, as of this writing the `applyAfterSolving` method is empty. When we wrote this class, we probably relied on the fact that `applyBeforeSolving` would also be called, which doesn’t strike me as a very good idea in hindsight: we’re not sure of what methods any given evolution scheme may call (this also makes `applyBeforeApplying` less safe than it seems).

The reason this worked is the way the `MixedScheme::step` method is implemented, as seen in the next listing. At each step, all boundary conditions are enforced first before and after applying the explicit part of the scheme, and then before and after solving for the implicit part.

In the case of the Dirichlet condition, this causes `applyAfterApplying` to fix any problem that `applyBeforeApplying` might have introduced and also ensures that `applyBeforeSolving` does the work even if `applyAfterSolving` doesn't.

If a boundary condition class implemented all of its methods correctly, this would be redundant; enforcing the condition just after the operations would be enough. However, we probably assumed that some conditions couldn't (or just didn't) implement all of them, and went for the safe option.

`MixedScheme::step` method, mark 2.

---

```
template <class Operator>
void MixedScheme<Operator>::step(array_type& a, Time t) {
    if (theta_ != 1.0) { // there is an explicit part
        for (Size i=0; i<bcs_.size(); i++)
            bcs_[i]->applyBeforeApplying(explicitPart_);
        a = explicitPart_.applyTo(a);
        for (Size i=0; i<bcs_.size(); i++)
            bcs_[i]->applyAfterApplying(a);
    }
    if (theta_ != 0.0) { // there is an implicit part
        for (Size i=0; i<bcs_.size(); i++)
            bcs_[i]->applyBeforeSolving(implicitPart_, a);
        implicitPart_.solveFor(a, a);
        for (Size i=0; i<bcs_.size(); i++)
            bcs_[i]->applyAfterSolving(a);
    }
}
```

---

If we ever set out to revamp this part of the library (not that likely, given the new framework) this is something we should get a look at. The problem to solve, unfortunately, is not an easy one: to call only the methods that are required by the scheme and supported by the boundary condition, we'd need to know the specific type of both.

Inverting the dependency and passing the scheme to the boundary condition, instead of the other way around, wouldn't work: the boundary condition wouldn't know if the scheme is calling `applyTo` or `solveFor`, and wouldn't be able to act inside the scheme's `step` method (as in `MixedScheme::step`, where the boundary conditions are enforced between the implicit and explicit parts of the step).

A mediator or some kind of visitor might work, but the complexity of the code would increase (especially if the number of conditions grow). Had they been available in C++, multimethods might have been the best choice; but even those might not be usable for multiple conditions. All in all, the current workaround of calling all method might be redundant but at least works correctly. The only alternative I can think of would be to require all conditions to be enforced after the operation, remove the `applyBefore...` methods and be done with it.

Enough with boundary conditions for now. With the pieces we have so far, we can take a payoff at  $t = T$  and evolve it back step by step to  $t = 0$ —if nothing happens in between, that is. But that's not often the case, which is my cue for the next subsection.

### 8.1.4 Step conditions

At any step (or in the middle of a step; we'll see later how we account for this) something might happen that changes the price of the asset; for instance, a right can be exercised or a coupon could be paid.

As opposed to boundary conditions, we called these step conditions. The name might not be the best; I can easily see the model as enforcing the “condition” that the asset price is the maximum between its continuation and the intrinsic value, but a coupon payment is harder to categorize in this way. In any case, the base class for all such conditions is shown in the listing below.

Interface of the `StepCondition` class template.

---

```
template <class array_type>
class StepCondition {
public:
    virtual ~StepCondition() {}
    virtual void applyTo(array_type& a, Time t) const = 0;
};
```

---

The `applyTo` method, which modifies the array of asset values in place, must also take care of checking that the condition applies at the given time; so, for instance, an American exercise will change the asset values regardless of the time, whereas a dividend payment will first check the passed time and bail out if it doesn't match the known payment time.

As you see, the base class is quite simple; but derived classes over-generalized and went South. In the next listing, you can see one such class, from which even simple conditions (like the one for American exercise) were derived. We could have implemented a simple `applyTo` method comparing the option values and the intrinsic values. Instead, we started adding constructors over the years so that instances could be built from either an existing array of intrinsic values, or a payoff, or some data that could describe the payoff. Of course these alternatives needed to be stored into different data members, so we added type erasure to the mix (Becker,2007), we defined an interface to retrieve intrinsic values, and we gave it an array-based and a payoff-based implementation.

Sketch of the `CurveDependentStepCondition` class template.

---

```

template <class array_type>
class CurveDependentStepCondition
    : public StepCondition<array_type> {
public:
    void applyTo(Array &a, Time) const {
        for (Size i = 0; i < a.size(); i++) {
            a[i] = applyToValue(a[i], getValue(a,i));
        }
    }
protected:
    CurveDependentStepCondition(Option::Type type,
                                Real strike);
    CurveDependentStepCondition(const Payoff *p);
    CurveDependentStepCondition(const array_type & a);

    class CurveWrapper {
        public:
            Real getValue(const array_type &a,
                          Size index) const = 0;
    };
    shared_ptr<CurveWrapper> curveItem_;

    Real getValue(const array_type &a, Size index) const {
        return curveItem_->getValue(a, index);
    }

    virtual Real applyToValue(Real, Real) const = 0;

    class ArrayWrapper : public CurveWrapper {
        array_type value_;
        public:
        ...
    };

    class PayoffWrapper : public CurveWrapper {
        shared_ptr<Payoff> payoff_;
        public:
        ...
    };
}

```

---

Ironically, we only ever used half of this. Existing engines use the array-based constructor and implementation, and the payoff-based inner class is not used anywhere in the library. Which is just as well; because, as I reviewed the code to write this chapter, I found out that its implementation is wrong. As I write, we're deprecating it in favor of a much simpler implementation of an American exercise condition that will just take an array of intrinsic values and implement `applyTo` accordingly. If you ever implement a step condition, I suggest you do the same.

A final note: the step condition is not applied inside the `step` method of the evolution scheme, as we'll see shortly. What this means is that there's no guarantee that the step condition won't break the boundary condition, or even the other way around. In practice, though, applying a step condition for a model that makes any sense will naturally enforce the chosen boundary condition.

And now, with all the basic pieces in place, let's start to put them together.

### Aside: look, Ma, no hands.

The faulty implementation of `PayoffWrapper` was added in February 2003, which probably makes this the longest-lived bug in the library (apart from those we still haven't found, of course). If I hadn't been looking at the code to write this chapter, the critter could have got old enough to drive.

Now, bugs happen; nothing shocking about this. What irks me, though, is that the problem would have been caught easily by a couple of unit tests. I and others usually try and review what goes in the library, and the code that gets contributed has often been used in the field and purged of obvious errors; but still I'm somewhat worried about the lack of coverage of old code in the test suite. If you have some time on your hands, please let me hear from you and I'll point you at our coverage reports.

#### 8.1.5 The `FiniteDifferenceModel` class

The `FiniteDifferenceModel` class, shown in the next listing, uses an evolution scheme to roll an asset back from a later time  $t_1$  to an earlier time  $t_2$ , taking into account any conditions we add to the model and stopping at any time we declare as mandatory (because something is supposed to happen at that point).

A word of warning: as I write this, I'm looking at this class after quite a few years and I realize that some things might have been done differently—not necessarily better, mind you, but there might be alternatives. After all this time, I can only guess at the reasons for such choices. While I describe the class, I'll point out the alternatives and write my guesses.

Interface of the `FiniteDifferenceModel` class template.

---

```
template<class Evolver>
class FiniteDifferenceModel {
public:
    typedef typename Evolver::traits traits;
    typedef typename traits::operator_type operator_type;
    typedef typename traits::array_type array_type;
    typedef typename traits::bc_set bc_set;
    typedef typename traits::condition_type condition_type;

    FiniteDifferenceModel(
        const Evolver& evolver,
        const std::vector<Time>& stoppingTimes =
            std::vector<Time>());
    FiniteDifferenceModel(
        const operator_type& L,
        const bc_set& bcs,
        const std::vector<Time>& stoppingTimes =
            std::vector<Time>());
    const Evolver& evolver() const{ return evolver_; }
    void rollback(array_type& a,
                 Time from,
                 Time to,
                 Size steps) {
        rollbackImpl(a,from,to,steps,
                     (const condition_type*) 0);
    }
    void rollback(array_type& a,
                 Time from,
                 Time to,
                 Size steps,
                 const condition_type& condition) {
        rollbackImpl(a,from,to,steps,&condition);
    }
private:
    void rollbackImpl(array_type& a,
                      Time from,
                      Time to,
                      Size steps,
                      const condition_type* condition);
    Evolver evolver_;
```

---

```
    std::vector<Time> stoppingTimes_;
```

};

---

I'll just mention quickly that the class takes the evolution scheme as a template argument (as in the code, I might call it *evolver* for brevity from now on) and extracts some types from it via traits; that's a technique you saw a few times already in earlier chapters.

So, first of all: the constructors. As you see, there's two of them. One, as you'd expect, takes an instance of the evolution scheme and a vector of stopping times (and, not surprisingly, stores copies of them). The second, instead, takes the raw ingredients and builds the evolver.

The reason for the second one might have been convenience, that is, to avoid writing the slightly redundant

```
FiniteDifferenceModel<SomeScheme>(SomeScheme(L, bcs), ts);
```

but I'm not sure that the convenience offsets having more code to maintain (which, by the way, only works for some evolvers and not for others; for instance, the base `MixedScheme` class doesn't have a constructor taking two arguments). This constructor might make more sense when using one of the typedefs provided by the library, such as `StandardFiniteDifferenceModel`, which hides the particular scheme used; but I'd argue that you should know what scheme you're using anyway, since they have different requirements in terms of grid spacing and time steps.<sup>8</sup>

The main logic of the class is coded in the `rollback` and `rollbackImpl` methods. The `rollback` method is the public interface and has two overloaded signatures: one takes the array of values to roll back, the initial and final times, and the number of steps to use, while the other also takes a `StepCondition` instance to use at each step. Both forward the call to the `rollbackImpl` method, which takes a raw pointer to step condition as its last argument; the pointer is null in the first case and contains the address of the condition in the second.

Now, this is somewhat unusual in the library: in most other cases, we'd just have a single public interface taking a `shared_ptr` and defining a null default value. I'm guessing that this arrangement was made to allow client code to simply create the step condition on the stack and pass it to the method, instead of going through the motions of creating a shared pointer (which, in fact, only makes sense if it's going to be shared; not passed, used and forgotten). Oh, and by the way, the use of a raw pointer in the private interface doesn't contradict the ubiquitous use of smart pointers everywhere in the library. We're not allocating a raw pointer here; we're passing the address of an object living on the stack (or inside a smart pointer, for that matter) and which can manage its lifetime very well, thank you.

One last remark on the interface: we're treating boundary conditions and step conditions differently, since the former are passed to the constructor of the model and the latter are passed to the `rollback`

---

<sup>8</sup>By the same argument, the `StandardFiniteDifferenceModel` typedef would be a suggestion of what scheme to use instead of a way to hide it.

method.<sup>9</sup> One reason could have been that this allows more easily to use different step conditions in different calls to `rollback`, that is, at different times; but that could be done inside the step condition itself, as it is passed the current time. The real generalization would have been to pass the boundary conditions to `rollback`, too, as it's currently not possible to change them at a given time in the calculation. This would have meant changing the interfaces of the evolvers, too: the boundary conditions should have been passed to their `step` method, instead of their constructors.

And finally, the implementation.

Implementation of the `FiniteDifferenceModel::rollbackImpl` method.

---

```
void FiniteDifferenceModel::rollbackImpl(
    array_type& a,
    Time from,
    Time to,
    Size steps,
    const condition_type* condition) {
    Time dt = (from-to)/steps, t = from;
    evolver_.setStep(dt);
    if (!stoppingTimes_.empty() && stoppingTimes_.back()==from) {
        if (condition)
            condition->applyTo(a,from);
    }
    for (Size i=0; i<steps; ++i, t -= dt) {
        Time now = t, next = t-dt;
        if (std::fabs(to-next) < std::sqrt(QL_EPSILON))
            next = to;
        bool hit = false;
        for (Integer j = stoppingTimes_.size()-1; j >= 0 ; --j) {
            if (next <= stoppingTimes_[j]
                && stoppingTimes_[j] < now) {
                hit = true;
                evolver_.setStep(now-stoppingTimes_[j]);
                evolver_.step(a,now);
                if (condition)
                    condition->applyTo(a,stoppingTimes_[j]);
                now = stoppingTimes_[j];
            }
        }
        if (hit) {
            if (now > next) {
                evolver_.setStep(now - next);
                evolver_.step(a,now);
            }
        }
    }
}
```

---

<sup>9</sup>Another difference is that we're using explicit sets of boundary conditions but just one step condition, which can be a composite of several ones if needed. I have no idea why.

---

```

        if (condition)
            condition->applyTo(a,next);
    }
    evolver_.setStep(dt);
} else {
    evolver_.step(a,now);
    if (condition)
        condition->applyTo(a, next);
}
}

```

---

We're going back from a later time `from` to an earlier time `to` in a given number of steps. This gives us a length `dt` for each step, which is set as the time step for the evolver. If `from` itself is a stopping time, we also apply the step condition (if one was passed; this goes for all further applications). Then we start going back step by step, each time going from the current time `t` to the next time, `t-dt`; we'll also take care that the last time is actually `to` and possibly snap it back to the correct value, because floating-point calculations might have got us almost there but not quite.

If there were no stopping times to hit, the implementation would be quite simple; in fact, the whole body of the loop would be the three lines inside the very last `else` clause:

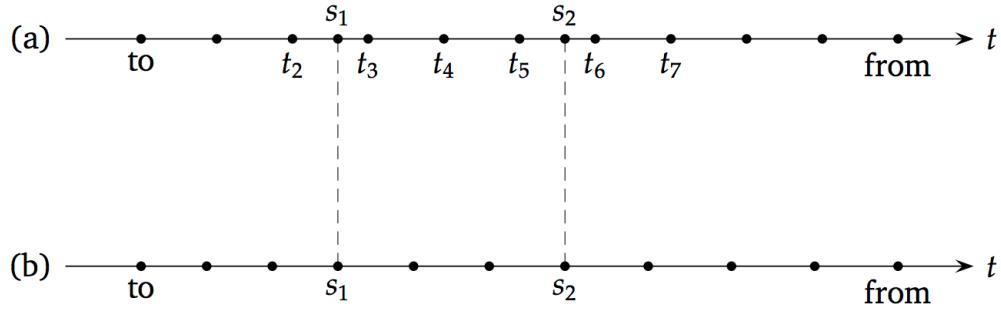
```

evolver_.step(a,now);
if (condition)
    condition->applyTo(a, next);

```

that is, we'd tell the evolver to roll back the values one step from `now`, and we'd possibly apply the condition at the target time `next`. This is what actually happens when there are no stopping times between `now` and `next`, so that the control variable `hit` remains false.

In case there are one or more stopping times in the middle of the step, things get a bit more complicated. Have a look at the upper half of the following figure, where I sketched a series of regular steps with two stopping times,  $s_1$  and  $s_2$ , falling in the middle of two of them.



Example of time grid for finite-difference models (a) and trees (b) including mandatory stopping times.

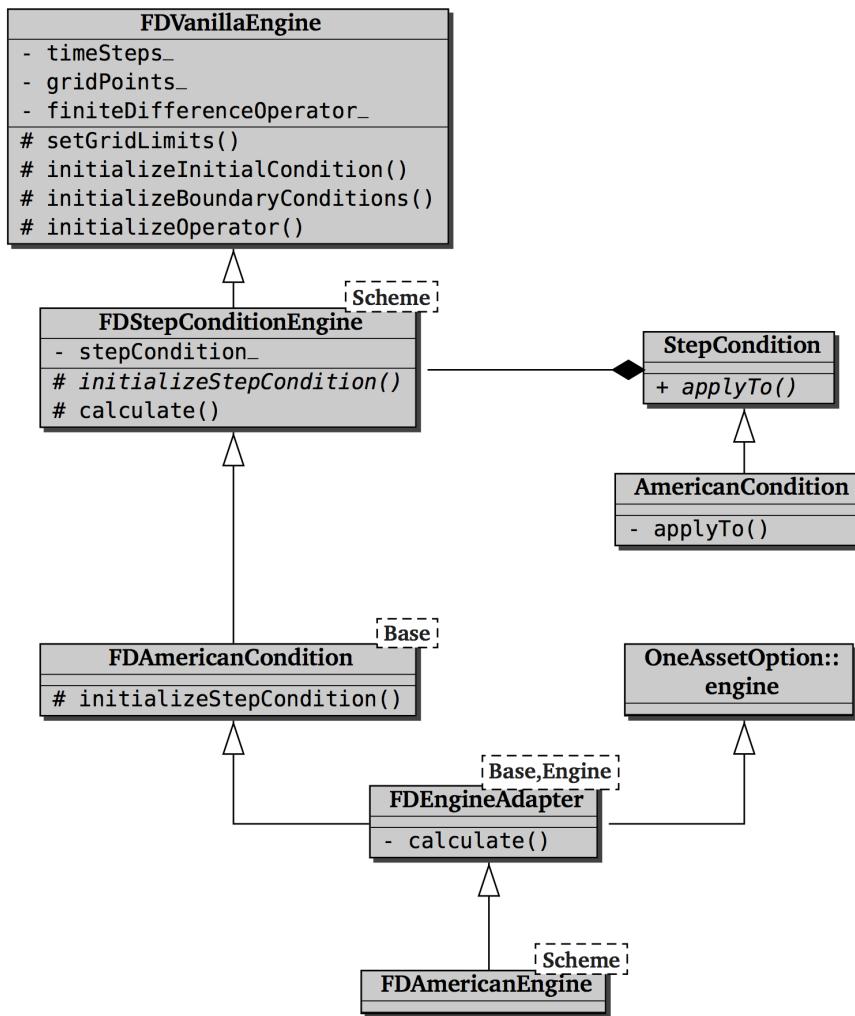
Let's say we have just made the step from  $t_7$  to  $t_6$  as described in the previous paragraph, and should now go back to  $t_5$ . However, the code will detect that it should stop at  $s_2$ , which is between now and next; therefore, it will set the control variable `hit` to true, it will set the step of the evolver to the smaller step  $t_6 - s_2$  that will bring it to the stopping time, and will perform the step (again, applying the condition afterwards). Another, similar change of step would happen if there were two or more stopping times in a single step; the code would then step from one to the other. Finally, the code will enter the `if (hit)` clause, which get things back to normal: it performs the remaining step  $s_2 - t_5$  and then resets the step of the evolver to the default value, so that it's ready for the next regular step to  $t_4$ .

As a final remark, let's go back to trees for a minute. In chapter 7, I mentioned in passing that the time grid of a lattice is built to include the mandatory stopping times. In this case, the grid would have been the one shown in the lower half of the previous figure. As you see, the stopping times  $s_1$  and  $s_2$  were made part of the grid by using slightly smaller steps between `to` and  $s_1$  and slightly larger steps between  $s_2$  and `from`.<sup>10</sup> I'm not aware of drawbacks in using either of the two methods, but experts might disagree so I'm ready to stand corrected. In any case, should you want to replicate the tree grid in a finite-difference model, you can do so by making several calls to `rollback` and going explicitly from stopping time to stopping time.

### 8.1.6 Example: American option

At this point, writing a finite-difference pricing engine should be just a matter of connecting the dots. Well, not quite. In this section, I'll sketch the implementation of an American-option engine in QuantLib, which is somewhat more complex than expected (as you can see for yourself from the following figure.)

<sup>10</sup>In this case, the change results in the same number of steps as the finite-difference grid. This is not always guaranteed; the grid for a tree might end up having a couple of steps more or less than the requested number.



Class diagram of `FDAMERICANENGINE`.

My reasons for doing this are twofold. On the one hand, I nearly got lost myself when I set out to write this chapter and went to read the code of the engine; so I thought it might be useful to put a map out here. On the other hand, this example will help me draw a comparison with the new (and more modular) framework.

Mind you, I'm not dissing the old implementation. The reason it got so complex was that we tried to abstract out reusable chunks of code, which makes perfect sense. The problem is that, although we didn't see it at the time, inheritance was probably the wrong way to do it.

Let's start with the `FDVanillaEngine` class, shown in the listing below. It can be used as a base class for both vanilla-option and dividend vanilla-option engines, which might explain why the name is

not as specific as, say, `FDVanillaOptionEngine`.<sup>11</sup>

Interface of the `FDVanillaEngine` class.

---

```

class FDVanillaEngine {
public:
    FDVanillaEngine(
        const shared_ptr<GeneralizedBlackScholesProcess>&,
        Size timeSteps, Size gridPoints,
        bool timeDependent = false);
    virtual ~FDVanillaEngine() {}
protected:
    virtual void setupArguments(
        const PricingEngine::arguments* ) const;
    virtual void setGridLimits() const;
    virtual void initializeInitialCondition() const;
    virtual void initializeBoundaryConditions() const;
    virtual void initializeOperator() const;

    shared_ptr<GeneralizedBlackScholesProcess> process_;
    Size timeSteps_, gridPoints_;
    bool timeDependent_;
    mutable Date exerciseDate_;
    mutable shared_ptr<Payoff> payoff_;
    mutable TridiagonalOperator finiteDifferenceOperator_;
    mutable SampledCurve intrinsicValues_;
    typedef BoundaryCondition<TridiagonalOperator> bc_type;
    mutable std::vector<shared_ptr<bc_type> > BCs_;

    virtual void setGridLimits(Real, Time) const;
    virtual Time getResidualTime() const;
    void ensureStrikeInGrid() const;
private:
    Size safeGridPoints(Size gridPoints,
                        Time residualTime) const;
};
```

---

This class builds most of the pieces required for a finite-difference model, based on the data passed to its constructor: a Black-Scholes process for the underlying, the number of desired time steps and grid points, and a flag that I'm going to ignore until the next subsection. Besides the passed inputs, the data members of the class include information to be retrieved from the instrument (that is, the exercise date and payoff) and the pieces of the model to be built: the differential operator,

---

<sup>11</sup>We might just have decided to shorten the name, though. I don't think anybody remembers after all these years.

the boundary conditions, and the array of initial values corresponding to the intrinsic values of the payoff. The latter array is stored in an instance of the `SampledCurve` class, which adds a few utility methods to the stored data.

The rest of the class interface is made of protected methods that builds and operates on the data members. I'll just go over them quickly: you can read their implementation in the library for more details.

First, the spectacularly misnamed `setupArguments` method does the opposite of its namesake in the `Instrument` class: it reads the required exercise and payoff information from the passed `arguments` structure and copies them into the corresponding data members of `FDVanillaEngine`.

The `setGridLimits` method determines and stores the minimum and maximum value of the logarithmic model grid, based on the variance of the passed process over the residual time of the option. The calculation enforces that the current value of the underlying is at the center of the grid, that the strike value is within its range, and that the number of its points is large enough.<sup>12</sup> The actual work is delegated to a number of other methods: an overloaded version of `setGridLimits`, `safeGridPoints`, and `ensureStrikeInGrid`.

The `initializeInitialCondition` method fills the array of intrinsic values by sampling the payoff on the newly specified grid; thus, it must be called after the `setGridLimits` method.

The `initializeBoundaryConditions` method, to be called as the next step, instantiates the lower and upper boundary conditions. They're both Neumann conditions, and the value of the derivative to be enforced is calculated numerically from the array of intrinsic values.

Finally, the `initializeOperator` method creates the tridiagonal operator based on the calculated grid and the stored process. Again, part of the work is delegated, this time to existing facilities outside the class.

All of these methods are declared as virtual, so that the default implementations can be overridden if needed. This is not optimal: in order to change any part of the logic one has to use inheritance, which introduces an extra concept just for customization and doesn't lend itself to different combinations of changes. A Strategy pattern would be better, and would also make some of the logic more reusable by other instruments.

All in all, though, the thing is manageable: see the `FDEuropeanEngine` class, shown in the next listing, which can be implemented in a reasonable amount of code. Its `calculate` method sets everything up by calling the appropriate methods from `FDVanillaEngine`, creates the model, starts from the intrinsic value of the option at maturity and rolls it back to the evaluation date. The value and a couple of Greeks are extracted by the corresponding methods of the `SampledCurve` class, and the theta is calculated from the relationship that the Black-Scholes equation imposes between it and the other results.<sup>13</sup>

---

<sup>12</sup>I'd note that the method might override the number of grid points passed by the user. In hindsight, I'm not sure that doing it silently is a good idea.

<sup>13</sup>By replacing the derivatives with the corresponding Greeks, the Black-Scholes equation says that  $\Theta + \frac{1}{2}\sigma^2 S^2 \Gamma + (r - q)S\Delta - rV = 0$ .

---

Implementation of the **FDEuropeanEngine** class.

---

```

template <template <class> class Scheme = CrankNicolson>
class FDEuropeanEngine : public OneAssetOption::engine,
                           public FDVanillaEngine {
public:
    FDEuropeanEngine(
        const shared_ptr<GeneralizedBlackScholesProcess>&,
        Size timeSteps=100, Size gridPoints=100,
        bool timeDependent = false);
private:
    mutable SampledCurve prices_;
    void calculate() const {
        setupArguments(&arguments_);
        setGridLimits();
        initializeInitialCondition();
        initializeOperator();
        initializeBoundaryConditions();

        FiniteDifferenceModel<Scheme<TridiagonalOperator> >
        model(finiteDifferenceOperator_, BCs_);

        prices_ = intrinsicValues_;

        model.rollback(prices_.values(), getResidualTime(),
                      0, timeSteps_);

        results_.value = prices_.valueAtCenter();
        results_.delta = prices_.firstDerivativeAtCenter();
        results_.gamma = prices_.secondDerivativeAtCenter();
        results_.theta = blackScholesTheta(process_,
                                           results_.value,
                                           results_.delta,
                                           results_.gamma);
    }
};
```

---

What with [the previous figure](#) then? How do we get three levels of inheritance between **FDVanillaEngine** and **FDAmericanEngine**? It was due to the desire to reuse whatever pieces of logic we could. As I said, the idea was correct: there are other options in the library that use part of this code, such as shout options, or options with discrete dividends. However, the architecture could have been simpler.

First, we have the **FDStepConditionEngine**, sketched in the following listing.

Sketch of the `FDStepConditionEngine` class.

---

```

template <template <class> class Scheme = CrankNicolson>
class FDStepConditionEngine : public FDVanillaEngine {
    public:
        FDStepConditionEngine(
            const shared_ptr<GeneralizedBlackScholesProcess>&,
            Size timeSteps, Size gridPoints,
            bool timeDependent = false);
    protected:
        // ...data members...
        virtual void initializeStepCondition() const = 0;
        virtual void calculate(PricingEngine::results*) const {
            OneAssetOption::results * results =
                dynamic_cast<OneAssetOption::results *>(r);
            setGridLimits();
            initializeInitialCondition();
            initializeOperator();
            initializeBoundaryConditions();
            initializeStepCondition();

            typedef /* ... */ model_type;

            prices_ = intrinsicValues_;
            controlPrices_ = intrinsicValues_;
            // ...more setup (operator, BC) for control...

            model_type model(operatorSet, bcSet);
            model.rollback(arraySet, getResidualTime(),
                          0.0, timeSteps_, conditionSet);

            results->value = prices_.valueAtCenter()
                - controlPrices_.valueAtCenter()
                + black.value();
            // same for Greeks
        }
    };

```

---

It represent a finite-difference engine in which a step condition is applied at each step of the calculation. In its `calculate` method, it implements the bulk of the pricing logic—and then some. First, it sets up the data members by calling the methods inherited from `FDVanillaEngine`, as well as an `initializeStepCondition` method that it declares as pure virtual and that derived classes must implement: it must create an instance of the `StepCondition` class appropriate for the given

engine. Then, it creates two arrays of values; the first for the option being priced, and the second for a European option that will be used as a control variate (this also requires to set up a corresponding operator, as well as a pricer object implementing the analytic Black formula). Finally, the model is created and used for both arrays, with the step condition being applied only to the first one, and the results are extracted and corrected for the control variate.

I don't have particular bones to pick with this class, except for the name, which is far too generic. I'll just add a note on the usage of control variates. We have already seen the technique in [chapter 6](#), where it was used to narrow down the width of the simulated price distribution; here it is used to improve the numerical accuracy. It is currently forced upon the user, since there's no flag allowing to enable or disable it; and it is relatively more costly than in Monte Carlo simulations (there, the path generation is the bulk of the computation and is shared between the option and the control; here, using it almost doubles the computational effort). The decision of whether it's worth using should be probably be left to the user. Also, we should use temporary variables for the control data instead of declaring other `mutable` data members; they're turning into a bad habit.

Next, the `FDAmericanCondition` class template, shown in the next listing.

It takes its base class as a template argument (in our case, it will be `FDVanillaEngine`) and provides the `initializeStepCondition` method, which returns an instance of the `AmericanCondition` class. Unfortunately, the name `FDAmericanCondition` is quite confusing: it suggests that the class is a step condition, rather than a building block for a pricing engine.

Sketch of the `FDAmericanEngine` class and its immediate base classes.

---

```
template <typename baseEngine>
class FDAmericanCondition : public baseEngine {
public:
    FDAmericanCondition(
        const shared_ptr<GeneralizedBlackScholesProcess>&,
        Size timeSteps = 100, Size gridPoints = 100,
        bool timeDependent = false);
protected:
    void initializeStepCondition() const;
};

template <typename base, typename engine>
class FDEngineAdapter : public base, public engine {
public:
    FDEngineAdapter(
        const shared_ptr<GeneralizedBlackScholesProcess>& p,
        Size timeSteps=100, Size gridPoints=100,
        bool timeDependent = false)
    : base(p, timeSteps, gridPoints, timeDependent) {
        this->registerWith(p);
}
```

```

private:
    void calculate() const {
        base::setupArguments(&(*this->arguments_));
        base::calculate(&(*this->results_));
    }
};

template <template <class> class Scheme = CrankNicolson>
class FDAmericanEngine
    : public FDEngineAdapter<
        FDAmericanCondition<
            FDStepConditionEngine<Scheme> >,
        OneAssetOption::engine> {

public:
    FDAmericanEngine(
        const shared_ptr<GeneralizedBlackScholesProcess>&,
        Size timeSteps=100, Size gridPoints=100,
        bool timeDependent = false);
}

```

---

The next to last step is the `FDEngineAdapter` class template, shown in the same listing, which connects an implementation and an interface by taking them as template arguments and inheriting from both: in this case, we'll have `FDAmericanCondition` as the implementation and `OneAssetOption::engine` as the interface. The class also provides a bit of glue code in its `calculate` method that satisfies the requirements of the engine interface by calling the methods of the implementation.

Finally, the `FDAmericanEngine` class inherits from `FDEngineAdapter` and specifies the classes to be used as bases.

The question is whether it is worth to increase the complexity of the hierarchy in order to reuse the bits of logic in the base classes. I'm not sure I have an answer, but I can show an alternate implementation and let you make the comparison on your own. If we let `FDAmericanEngine` inherit directly from `FDStepConditionEngine` and `OneAssetOption::engine`, and if we move into this class the code from both `FDAmericanCondition` and `FDEngineAdapter` (that we can remove afterwards), we obtain the implementation in the listing below.

Alternate implementation of the `FDAmericanEngine` class.

---

```
template <template <class> class Scheme = CrankNicolson>
class FDAmericanEngine
: public FDStepConditionEngine<Scheme>,
public OneAssetOption::engine {
    typedef FDStepConditionEngine<Scheme> fd_engine;
public:
    FDAmericanEngine(
        const shared_ptr<GeneralizedBlackScholesProcess>& p,
        Size timeSteps=100, Size gridPoints=100,
        bool timeDependent = false)
    : fd_engine(p, timeSteps, gridPoints, timeDependent) {
        this->registerWith(p);
    }
protected:
    void initializeStepCondition() const;
    void calculate() const {
        fd_engine::setupArguments(&(this->arguments_));
        fd_engine::calculate(&(this->results_));
    }
};
```

---

My personal opinion? I tend to lean towards simplicity in my old age. The code to be replicated would be little, and the number of classes that reuse it is not large (about half a dozen in the current version of the library). Moreover, the classes that we'd remove (`FDAmericanCondition` and `FDEngineAdapter`) don't really model a concept in the domain, so I'd let them go without any qualms. Too much reuse without a proper abstraction might be a thing, after all.

A final note: as you can see, in this framework there are no high-level classes encapsulating generic model behavior, such as `McSimulation` for Monte Carlo (see section 6.3). Whatever logic we had here was written in classes meant for a specific instrument—in this case, plain options in a Black-Scholes-Merton model.

### 8.1.7 Time-dependent operators

So far, I ignored the possibility that the parameters of the model depend on time. However, the basic classes in the framework need little modification to cover that case; the additional code is shown in the next listing.

Code to support time dependence in finite-differences classes.

---

```

class TridiagonalOperator {
    public:
        class TimeSetter {
            public:
                virtual ~TimeSetter() {}
                virtual void setTime(Time t,
                                      TridiagonalOperator&) const = 0;
            };
            ...
            bool isTimeDependent() const { return !timeSetter_; }
            void setTime(Time t) {
                if (timeSetter_)
                    timeSetter_->setTime(t, *this);
            }
        protected:
            ...
            shared_ptr<TimeSetter> timeSetter_;
    };

template <class Operator>
class BoundaryCondition {
    public:
        ...
        virtual void setTime(Time t) = 0;
};

template <class Operator>
void MixedScheme<Operator>::step(array_type& a, Time t) {
    for (Size i=0; i<bcs_.size(); i++)
        bcs_[i]->setTime(t);
    if (theta_!=1.0) { // there is an explicit part
        if (L_.isTimeDependent()) {
            L_.setTime(t);
            explicitPart_ = I_-((1.0-theta_) * dt_)*L_;
        }
        // continue as before
    }
    if (theta_!=0.0) {
        // the same for the implicit part
    }
}

```

---

The `TridiagonalOperator` class gains an inner class, `TimeSetter`, and a data member holding an instance (possibly null) of the same. The interface of the inner class, apart from the destructor, consists of the single method `setTime` that takes a reference to the operator and modifies its elements according to the current time (also passed). Of course, the method is declared as pure virtual; the actual logic will be implemented in derived classes. In what can be seen as an implementation of the Strategy pattern, the `setTime` method of `TridiagonalOperator` works by delegating the job to the stored setter, if any.

The `BoundaryCondition` class doesn't follow the same pattern; instead, it simply declares a virtual `setTime` method that is supposed to perform any required setup. In the simple Dirichlet and Neumann conditions available from the library, the method does nothing.

Finally, the code of the evolution schemes needs to be modified to take into account the possibility of time dependence. As an example, the listing shows the modified `step` method from the `MixedScheme` class template; you can compare it with the version in [section 8.1.3](#). First, it calls the `setTime` method on all boundary conditions; then, if the operator is time dependent, it calls its `setTime` method and recomputes the explicit and implicit operators  $I - (1 - \theta) \cdot \Delta t \cdot L$  and  $I + \theta \cdot \Delta t \cdot L$  as required. The remaining calculations are performed as before.

Unfortunately, the available implementations of actual time-dependent operators are not as simple as the modifications required in the basic classes. The listing below shows part of the code used to implement a Black-Scholes-Merton operator with time-dependent coefficients.

The `BSMTermOperator` class and related classes.

---

```

class PdeSecondOrderParabolic {
    public:
        virtual ~PdeSecondOrderParabolic() {}
        virtual Real diffusion(Time t, Real x) const = 0;
        virtual Real drift(Time t, Real x) const = 0;
        virtual Real discount(Time t, Real x) const = 0;
        virtual void generateOperator(Time t,
                                       const TransformedGrid& tg,
                                       TridiagonalOperator& L) const {
            for (Size i=1; i < tg.size() - 1; i++) {
                Real sigma = diffusion(t, tg.grid(i));
                Real nu = drift(t, tg.grid(i));
                Real r = discount(t, tg.grid(i));
                Real sigma2 = sigma * sigma;

                Real pd = -(sigma2/tg.dxm(i)-nu)/ tg.dx(i);
                Real pu = -(sigma2/tg.dxp(i)+nu)/ tg.dx(i);
                Real pm = sigma2/(tg.dxm(i) * tg.dxp(i))+r;
                L.setMidRow(i, pd,pm,pu);
            }
        }
}
```

```

};

class PdeBSM : public PdeSecondOrderParabolic {
public:
    PdeBSM(const shared_ptr<GeneralizedBlackScholesProcess>&);
    ...
};

template <class PdeClass>
class PdeOperator : public TridiagonalOperator {
public:
    PdeOperator(const Array& grid, const PdeClass& pde)
        : TridiagonalOperator(grid.size()) {
        timeSetter_ =
            shared_ptr<GenericTimeSetter<PdeClass> >(
                new GenericTimeSetter<PdeClass>(grid, pde));
    }
};

typedef PdeOperator<PdeBSM> BSMTermOperator;

```

---

Like for the engine I've shown in the previous example, several bits of behavior were put in different classes in order to reuse them. For instance, the `PdeSecondOrderParabolic` class can be used for partial differential equations (PDE) of the form

$$\frac{\partial f}{\partial t} = -\nu(t, x) \frac{\partial f}{\partial x} - \frac{1}{2} \sigma^2(t, x) \frac{\partial^2 f}{\partial x^2} + r(t, x) f$$

and defines a `generateOperator` method that calculates the coefficients of the corresponding tridiagonal operator, given a grid and a time; the  $\nu$ ,  $\sigma$  and  $r$  coefficients above are provided by inheriting from the class and overriding the `drift`, `diffusion` and `discount` methods, respectively. As you might have noticed, `discount` is a misnomer; the coefficient actually corresponds to the instantaneous risk-free rate.<sup>14</sup> In our case, the derived class `PdeBSM` specifies the coefficients for the Black-Scholes-Merton process; the methods above, not shown for brevity, are implemented by fetching the corresponding information from the passed process.

The `PdeOperator` class template, shown in the same listing, inherits from `TridiagonalOperator`. It takes as template argument a class like `PdeBSM`; that is, one that provides a `generateOperator` method with the correct semantics. The constructor uses an instance of such class, together with a given grid, to build a time setter. Once the setter is stored, the `PdeOperator` instance can be safely sliced and passed around as a `TridiagonalOperator` instance. The setter is an instance of the

<sup>14</sup>Come to think of it, `generateOperator` is not the correct name, either. The method doesn't create an operator; it fills, or updates, the coefficients of an existing one.

`GenericTimeSetter` class template, not shown here, that implements its required `setTime` method by passing the given time, the stored grid and the operator to the `generateOperator` method of the stored PDE class.

Finally, by instantiating `PdeOperator` with the `PdeBSM` class we get a time-dependent Black-Scholes-Merton operator to which we can give a name by means of a `typedef`.

Again, I'm not sure that this is the right balance between readability and reusability (and to be fair, I'm not sure it isn't, either). Like in the previous example, you could simplify the hierarchy by putting the whole thing in a single `BSMTimeSetter` class. Its constructor would store the grid and a process, and its `setTime` method would contain the code which is currently split between the several classes in the previous listing. All in all, I think we're short of actual evidence about which version is best.

\* \* \*

There are lots of other things in the framework that I could describe. There's a `PdeShortRate` class, which works as `PdeBSM` for one-factor short-rate models. There's an `OperatorTraits` class used to define a few types used around the framework. There are others I don't even remember.

However, I'll leave them for you to explore—with the caveat that, in time, they might be deprecated and disappear. It's time to go onwards and have a look at the new finite-differences framework.

## 8.2 The new framework

Since this is the last chapter before the appendices, I'll grab my last chance to describe some code from a top-down perspective. Instead of starting from the basic framework components and work my way up to higher-level classes, I'll show you what a completed finite-difference engine looks like in the new framework. It's the `FdBlackScholesVanillaEngine` class; its declaration and the implementation of its `calculate` method are shown in the listing below.

Interface of the `FdBlackScholesVanillaEngine` class.

---

```

class FdBlackScholesVanillaEngine
    : public DividendVanillaOption::engine {
public:
    FdBlackScholesVanillaEngine(
        const shared_ptr<GeneralizedBlackScholesProcess>&,
        Size tGrid=100, Size xGrid=100, Size dampingSteps=0,
        const FdmSchemeDesc& schemeDesc=FdmSchemeDesc::Douglas(),
        bool localVol=false,
        Real illegalLocalVolOverwrite=-Null<Real>());
void calculate() const;

```

```

private:
    // data members, not shown
};

void FdBlackScholesVanillaEngine::calculate() const {
    shared_ptr<StrikedTypePayoff> payoff =
        dynamic_pointer_cast<StrikedTypePayoff>(
            arguments_.payoff);
    Time maturity = process_->time(
        arguments_.exercise->lastDate());
    shared_ptr<Fdm1dMesher> equityMesher(
        new FdmBlackScholesMesher(
            xGrid_, process_, maturity, payoff->strike(),
            Null<Real>(), Null<Real>(), 0.0001, 1.5,
            std::pair<Real, Real>(payoff->strike(), 0.1)));
    shared_ptr<FdmMesher> mesher(
        new FdmMesherComposite(equityMesher));
    shared_ptr<FdmInnerValueCalculator> calculator(
        new FdmLogInnerValue(payoff, mesher, 0));
    shared_ptr<FdmStepConditionComposite> conditions =
        FdmStepConditionComposite::vanillaComposite(
            arguments_.cashFlow, arguments_.exercise,
            mesher, calculator,
            process_->riskFreeRate()->referenceDate(),
            process_->riskFreeRate()->dayCounter());
    FdmBoundaryConditionSet boundaries;
    FdmSolverDesc solverDesc = {
        mesher, boundaries, conditions, calculator,
        maturity, tGrid_, dampingSteps_
    };
    shared_ptr<FdmBlackScholesSolver> solver(
        new FdmBlackScholesSolver(
            Handle<GeneralizedBlackScholesProcess>(process_),
            payoff->strike(), solverDesc, schemeDesc_,
            localVol_, illegalLocalVolOverwrite_));
    Real spot = process_->x0();
}

```

---

```

    results_.value = solver->valueAt(spot);
    // other Greeks
}

```

---

The difference from, say, the `FDAmericanEngine` class in [an earlier section](#) is straightforward enough. Instead of inheriting functionality from base classes, the new-style engine performs its required calculations by coordinating a number of components (such as meshers and calculators and conditions, oh my) that, at least in principle, can be reused by other engines. In the rest of this section, I'll examine them to see how they work and interact.

### 8.2.1 Meshers

The first component to be built is an instance of the `Fdm1dMesher` class, shown in the next listing; or more precisely, an instance of its derived `FdmBlackScholesMesher` class. Each instance of `Fdm1dMesher` stores the discretization of one dimension of the problem domain; in this case, we need only one modeling the desired range for the underlying value.

Interface of the `Fdm1dMesher` class.

---

```

class Fdm1dMesher {
public:
    Fdm1dMesher(Size size);
    Size size() const;
    Real dplus(Size index) const;
    Real dminus(Size index) const;
    Real location(Size index) const;
    const std::vector<Real>& locations();
protected:
    std::vector<Real> locations_;
    std::vector<Real> dplus_, dminus_;
};

```

---

The base class doesn't have behavior, apart from a few inspectors. It stores a vector named `locations_` that contains a set of points  $\{x_0, x_1, \dots, x_{n-1}\}$  discretizing the domain for  $x$ ; and for convenience, it also precomputes two other vectors `dplus_` and `dminus_` whose  $i$ -th elements contain  $(x_{i+1} - x_i)$  and  $(x_i - x_{i-1})$ , respectively. The vectors are declared as `protected`, since filling them is left as an exercise to the constructors of derived classes.

Even if its instances are passed around as `shared_ptr`s, the class is not polymorphic; none of the methods in the interface are virtual.<sup>15</sup> In fact, there seems to be no reason for passing instances

---

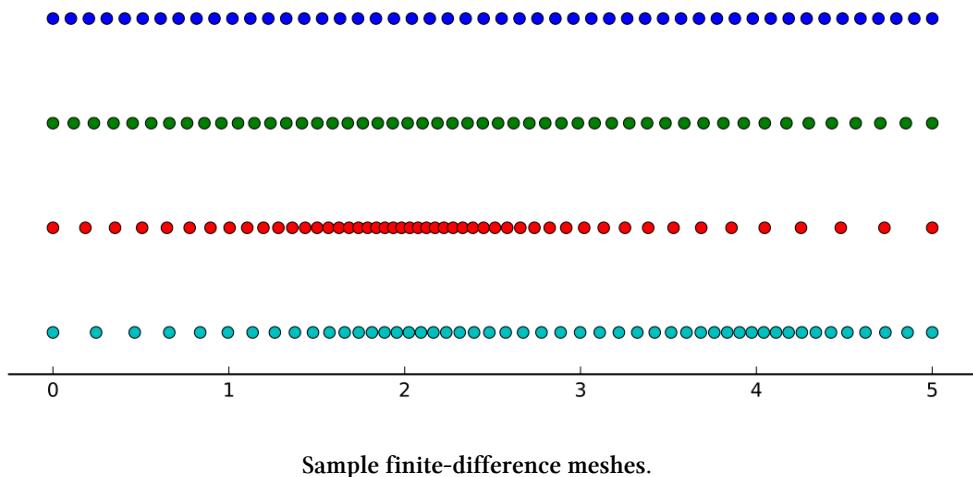
<sup>15</sup>The class doesn't even have a virtual destructor, which might cause undefined behavior in case a pointer to the base class is deleted as such. In this case, we're safe thanks to the constructor of `shared_ptr`, which stores the correct deleter for the specific pointer type it was passed.

around as pointers, except maybe to avoid copying; and even this might not matter (or wouldn't even hold if you're compiling in C++11 mode). In hindsight, the `Fdm1dMesher` class could have been written as a class not meant for inheritance, with its derived classes being replaced by functions building and returning actual `Fdm1dMesher` instances.

The `FdmBlackScholesMesher` constructor, that I won't show here, performs a few calculations to determine the grid boundaries based on the current value of the underlying, its expected variance from now to the maturity of the option, and possibly some external constraints (such as barriers, for instance). It is meant to be used not only in this engine, but also in others for which the underlying follows the same process. The calculations are similar to those implemented in the old framework by the `FDVanillaEngine` class, except that the new code allows you to tune some of the parameters hard-coded in the old one; and like in the old framework, the grid is actually for the logarithm of the underlying.

After calculating the boundaries, the constructor delegates the actual construction of the grid to one of two generic utility classes, both inherited from `Fdm1dMesher`, and then copies their data; if it was a function as mentioned above, it could just return the resulting mesh instead.

The simpler of the two classes is named `Uniform1dMesher`, and does what its name advertises: it creates an equally-spaced grid between two given boundaries and with the given number of points. The other class is used when there are one or more points of particular interest in the grid, such as the option strike, and is named `Concentrating1dMesher`. Its constructors build a grid with a given number of points, but concentrates them around its so-called critical points. For each of them the constructor takes the value of the point, another value to tune the density of the points around it, and a boolean that specifies whether the critical point should actually belong to the mesh. The figure below shows some possible resulting meshes: from top to bottom, a uniform mesh; a concentrating mesh with a critical point at  $x = 2$ ; a concentrating mesh with a critical point at  $x = 2$  and tuned for a higher point density around it; and a concentrating mesh with critical points at  $x = 2$  and  $x = 4$ . All meshes have the same number of points.



The full multi-dimensional mesh for a finite-difference model is represented by the abstract `FdmMesher` class, shown in the following listing. In the engine at the beginning of the section, we're actually instantiating the derived `FdmMesherComposite` class (also shown here), which builds the full mesh by composing a 1-D mesh for every dimension of the problem; in this particular case, only the one we built for the underlying value.

Interface of the `FdmMesher` and `FdmMesherComposite` classes.

---

```

class FdmMesher {
    public:
        FdmMesher(const shared_ptr<FdmLinearOpLayout>&);

        virtual ~FdmMesher() {}

        virtual Real dplus(const FdmLinearOpIterator& iter,
                           Size direction) const = 0;
        virtual Real dminus(const FdmLinearOpIterator& iter,
                           Size direction) const = 0;
        virtual Real location(const FdmLinearOpIterator& iter,
                             Size direction) const = 0;
        virtual Array locations(Size direction) const = 0;

        const shared_ptr<FdmLinearOpLayout>& layout() const;
};

class FdmMesherComposite : public FdmMesher {
    public:
        explicit FdmMesherComposite(
            const shared_ptr<Fdm1dMesher>& mesher);
        FdmMesherComposite(
            const shared_ptr<Fdm1dMesher>& m1,
            const shared_ptr<Fdm1dMesher>& m2);
        // ... constructors for up to 4 meshers ...
        explicit FdmMesherComposite(
            const std::vector<shared_ptr<Fdm1dMesher> >&);

        Real dplus(const FdmLinearOpIterator& iter,
                   Size direction) const;
        Real dminus(const FdmLinearOpIterator& iter,
                    Size direction) const;
        Real location(const FdmLinearOpIterator& iter,
                      Size direction) const;
        Array locations(Size direction) const;
};

```

---

Unfortunately, I can't explain the interface of `FdmMesher` in full without going a bit down a rabbit hole. The idea is that, for any given point in the full mesh (specified through some kind of iterator) its methods can give the location and the increments `dplus` and `dminus` along any of the dimensions. But to understand it more clearly, we have first to look at the representation of multi-dimensional differential operators and arrays.

## 8.2.2 Operators

The interface of the `FdmLinearOp` class is shown in the next listing, and its main feature is an `apply` method that returns the result of applying the operator to a given array; you might remember a similar `applyTo` method defined by the old-style operators.<sup>16</sup>

Interface of the `FdmLinearOp` class and its helper classes.

---

```

class FdmLinearOp {
    public:
        typedef Array array_type;
        virtual ~FdmLinearOp() { }
        virtual Disposable<array_type> apply(
            const array_type& r) const = 0;
        virtual Disposable<SparseMatrix> toMatrix() const = 0;
    };

    class FdmLinearOpIterator {
        public:
            explicit FdmLinearOpIterator(const std::vector<Size>& dim);
            explicit FdmLinearOpIterator(Size index = 0);
            FdmLinearOpIterator(const std::vector<Size>& dim,
                const std::vector<Size>& coordinates,
                Size index)

            void operator++();
            bool operator!=(const FdmLinearOpIterator&);
    };

    class FdmLinearOpLayout {
        public:
            FdmLinearOpLayout(const std::vector<Size>& dim);

            FdmLinearOpIterator begin() const;
            FdmLinearOpIterator end() const;
    };
}
```

---

<sup>16</sup>I'll gloss over the `toMatrix` method, which returns some kind of representation of the operator as a matrix and seems to be only used for inspection.

```

    Size index(const std::vector<Size>& coordinates) const;
    Size neighbourhood(const FdmLinearOpIterator& iterator,
                      Size i, Integer offset) const;
    Size neighbourhood(const FdmLinearOpIterator& iterator,
                      Size i1, Integer offset1,
                      Size i2, Integer offset2) const;
};


```

---

However, you probably noticed that the `array_type` used in the interface is defined as a `typedef` to `Array`, which implements a one-dimensional array. What of my claim that we can also cover multi-dimensional operators, then? Well, the framework maps the elements of a multi-dimensional array to the elements of a one-dimensional one by flattening the former and turning it into the latter.

As an example, let's say that we have a three-dimensional array  $a_{i,j,k}$  with  $0 \leq i < M$ ,  $0 \leq j < N$ , and  $0 \leq k < P$ . We'll map it into a one-dimensional sequence via a lexicographic ordering of the elements. We start from  $a_{0,0,0}$  as the first element, followed by  $a_{1,0,0}$  up to  $a_{M-1,0,0}$ . Then we increase the second index and reset the first, obtaining  $a_{0,1,0}$  and following with  $a_{1,1,0}$  to  $a_{M-1,1,0}$ . The next element is  $a_{0,2,0}$ , and this repeats until we get to  $a_{M-1,N-1,0}$ , at which point we follow with  $a_{0,0,1}$ . You got the gist; we go through the whole thing again to  $a_{M-1,N-1,1}$ , then  $a_{0,0,2}$  and so on until we finish with  $a_{M-1,N-1,P-1}$ .

The `FdmLinearOpIterator` and `FdmLinearOpLayout` classes, whose interfaces were also shown in the previous listing, cooperate to implement the logic above. And yes, even though both classes are named after the `FdmLinearOp` class, they actually work on the arrays to which operators are applied.

Instances of the `FdmLinearOpIterator` keep track of a tuple of indices into the several dimensions of an array (like the  $(i, j, k)$  tuples above) and of the corresponding single index into the flattened array; to do that, they must also store the sizes of the array along each dimension ( $M$ ,  $N$  and  $P$  in the example).

Depending on the constructor you'll call, you can pass explicitly all the state or just part of it. The constructor taking only a vector of dimensions stores the sizes of the array and implicitly initializes all the indices to 0; that is, it returns the iterator pointing to the beginning of the array. The constructor taking a single index builds a kind of sentinel value; it cannot be incremented, because it doesn't know the size of the array, but when it is passed the total number of elements of the array it returns an `end` iterator to which others can be compared to end a loop, as in:

```
std::vector<Size> dims(...); // { M, N, P }
FdmLinearOpIterator begin(dims);
FdmLinearOpIterator end(M*N*P);
for (FdmLinearOpIterator i=begin; i!=end; ++i)
    ... // do something
```

Finally, the third constructor takes the dimensions and the whole set of indices, and thus allows you to create an iterator pointing to any element of the array. For performance reasons (I guess) it leaves to the caller to check that the parameters are consistent, that is, that the set of indices into the array actually corresponds to the single index into the flattened array.

A `FdmLinearOpIterator` instance is not really an iterator in the C++ sense, since it doesn't implement the whole required interface; in particular, there's no dereference operator returning the value it points to. What it does implement is `operator++`, which increments the iterator, and `operator!=`, which compares two iterators.<sup>17</sup> the two methods are shown in the listing below. To round up their interface, iterators also declare the `coordinates` inspector, which returns the set of indices into the array, and `index`, returning the corresponding single index.

Partial implementation of the `FdmLinearOpIterator` class.

---

```
void FdmLinearOpIterator::operator++() {
    ++index_;
    for (Size i=0; i < dim_.size(); ++i) {
        if (++coordinates_[i] == dim_[i])
            coordinates_[i] = 0;
        else
            break;
    }
}

bool FdmLinearOpIterator::operator!=(
    const FdmLinearOpIterator& iterator) {
    return index_ != iterator.index_;
}
```

---

The increment operator literally implements the iterating strategy I described in the previous page. It starts by incrementing the first index; if doing so causes it to reach the size of the first dimension, it resets the index to 0 and repeats both the increment and check on the next dimension; otherwise, it breaks out of the loop. Thus, for instance, calling `operator++` with the indices set to  $(M - 1, N - 1, 2)$  will increment the first and get to  $(M, N - 1, 2)$ , then reset it and increment the second to  $(0, N, 2)$ , then reset the second too and increment the third to  $(0, 0, 3)$ , then finally exit. Calling it with the indices set to  $(2, 0, 0)$ , instead, will just increment the first to  $(3, 0, 0)$  and exit. In any case, the single index into the flattened array is also incremented so that it stays in sync with the other indices.

<sup>17</sup>Providing only `operator!=` and not `operator==` is enough to support the `i != end` idiom used in for loops, but might be a tad extreme.

Finally, `operator!=` compares two iterators by comparing their two indices into the flattened array (also known as the indices for which I should probably get a shorter name at this point). This allows iterators to work as sentinels even if they don't have a full set of indices and can't be incremented.

The `FdmLinearOpLayout` class (whose interface we have seen above, and whose implementation is shown in part in the next listing) provides some utilities for working with iterators based on a given array layout, or set of dimensions.

---

**Partial implementation of the `FdmLinearOpLayout` class.**

---

```

FdmLinearOpLayout::FdmLinearOpLayout(
    const std::vector<Size>& dim)
: dim_(dim), spacing_(dim.size()) {
    spacing_[0] = 1;
    std::partial_sum(dim.begin(), dim.end()-1,
        spacing_.begin()+1, std::multiplies<Size>());
    size_ = spacing_.back()*dim.back();
}

FdmLinearOpIterator FdmLinearOpLayout::begin() const {
    return FdmLinearOpIterator(dim_);
}

FdmLinearOpIterator FdmLinearOpLayout::end() const {
    return FdmLinearOpIterator(size_);
}

Size FdmLinearOpLayout::index(
    const std::vector<Size>& coordinates) const {
    return std::inner_product(coordinates.begin(),
        coordinates.end(),
        spacing_.begin(), Size(0));
}

```

---

Most of its calculations are based on what it calls *spacing* between the elements of the array along the various directions. I'll use the same example as before, and consider a three-dimensional array with sizes  $(M, N, P)$  along the three dimensions. If we take two elements  $(i, j, k)$  and  $(i+1, j, k)$ , they occupy consecutive places on the one-dimensional flattened array (remember that we increment the first index first while enumerating the elements) so we say that the spacing between them is 1. If we take two elements  $(i, j, k)$  and  $(i, j+1, k)$ , their positions in the flattened array are separated by a whole turn of the first index: enumerating the elements between them will cause you to go through  $(i+1, j, k)$ ,  $(i+2, j, k)$  and so on up to  $(M-1, j, k)$ , then to  $(0, j+1, k)$ , and then again through  $(1, j+1, k)$ ,  $(2, j+1, k)$  up to  $(i, j+1, k)$ . A bit of thinking will convince you that the spacing between the two elements is  $M$ , and a similar argument will find a spacing of  $M \times N$  between elements  $(i, j, k)$  and  $(i, j, k+1)$ .

The constructor of a `FdmLinearOpLayout` instance takes a general set of sizes  $(n_1, \dots, n_N)$  and calculates the spacings  $(1, n_1, n_1 \times n_2, \dots, \prod_1^{N-1} n_i)$  as well as the total size  $\prod_1^N n_i$ ; the calculations is written in STL speak, but it shouldn't be difficult to map it to the formulas.

Given the sizes and the spacings, the class can implement a number of methods that let you (and library code) work with iterators. The `begin` and `end` methods return the corresponding iterators, so that once you have a layout you no longer need to care about sentinels and iterator constructors and loops can be written as:

```
std::vector<Size> dims(...);
FdmLinearOpLayout l(dims);
for (FdmLinearOpIterator i=l.begin(); i!=l.end(); ++i)
    ... // do something
```

The `index` method provides a conversion between a set of indices and the single index into the flattened array. Other methods, such as `neighborhood`, work at a slightly higher level; they return the index (or the iterator) of the element you get by starting from a given element, whose position is described by an iterator, and moving a given number of steps in a given dimension. An overload that moves in two directions is also available for convenience, even though it's equivalent to two consecutive calls to the first version.

At this point, we can finally make sense of the interface of the `FdmMesher` class I've shown earlier. The methods `dplus`, `dminus` and `location` all take as first argument an iterator to specify a point in the mesh;<sup>18</sup> the second argument specifies the direction along which we're moving. The `layout` method returns an instance of `FdmLinearOpLayout` that can be used to create iterators.

And now, some operators. As for the old framework, the most basic would be those that represent the first and second derivative along one of the axes. In this case, "basic" doesn't necessarily mean "simple"; the formula for a three-point first derivative when the grid  $\{x_1, \dots, x_i, \dots, x_N\}$  is not equally spaced is {#formula-nonuniform-derivative}

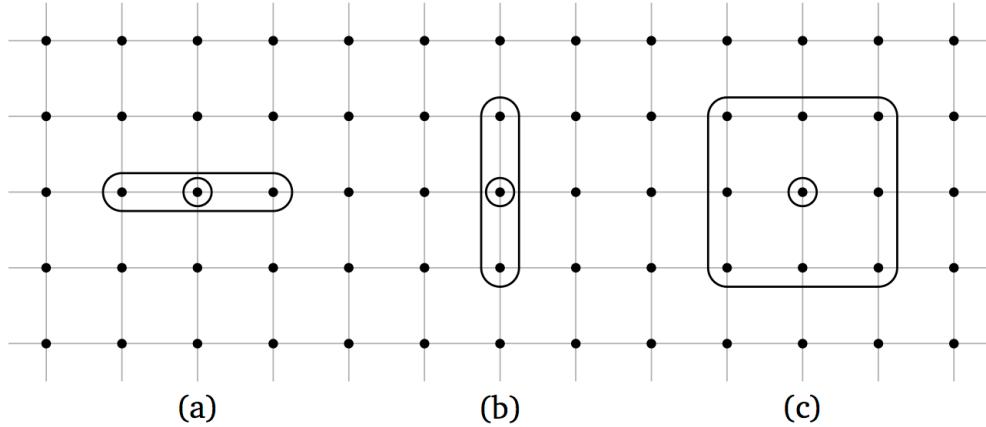
$$\frac{\partial f}{\partial x}(x_i) \approx -\frac{h_+}{h_-(h_- + h_+)} f(x_{i-1}) + \frac{h_+ - h_-}{h_+ h_-} f(x_i) + \frac{h_-}{h_+(h_- + h_+)} f(x_{i+1})$$

where  $h_- = x_i - x_{i-1}$  and  $h_+ = x_{i+1} - x_i$ , and the formula for the second derivative is just as long (if you're interested, you can find the derivation of the formulas in Bowen and Smith ([Bowen and Smith, 2005](#)); preprints of the paper are available on the web). Furthermore, even though I've used the notation  $x_i$  to denote the  $i$ -th point along the relevant dimension, the points  $x_{i-1}$ ,  $x_i$  and  $x_{i+1}$  in the formula might actually be  $x_{i,j,k-1,l}$ ,  $x_{i,j,k,l}$  and  $x_{i,j,k+1,l}$  if we're using a multi-dimensional array.

In any case, we'll be using a three-point stencil such as the ones shown for two different directions in plots (a) and (b) of the following figure.

---

<sup>18</sup>The points in the mesh are in one-to-one correspondence with the elements of the array containing the values of the instrument being priced on the mesh, thus an iterator can denote either one.



Stencils used for basic finite-difference derivatives.

Thus, both first- and second-derivative operators can be represented as instantiations of the same underlying class, called `TripleBandLinearOp` and shown in the next listing.

Partial implementation of the `TripleBandLinearOp` class.

```

class TripleBandLinearOp : public FdmLinearOp {
    public:
        TripleBandLinearOp(Size direction,
                            const shared_ptr<FdmMesher>& mesher);

        Array apply(const Array& r) const;
        Array solve_splitting(const Array& r, Real a,
                             Real b = 1.0) const;

        TripleBandLinearOp mult(const Array& u) const;
        TripleBandLinearOp multR(const Array& u) const;
        TripleBandLinearOp add(const TripleBandLinearOp& m) const;
        TripleBandLinearOp add(const Array& u) const;

        void axpyb(const Array& a, const TripleBandLinearOp& x,
                    const TripleBandLinearOp& y, const Array& b);
    protected:
        TripleBandLinearOp() {}
        Size direction_;
        shared_array<Size> i0_, i2_;
        shared_array<Size> reverseIndex_;
        shared_array<Real> lower_, diag_, upper_;
        shared_ptr<FdmMesher> mesher_;

```

```

};

TripleBandLinearOp::TripleBandLinearOp(
    Size direction,
    const shared_ptr<FdmMesher>& mesher) : /* ... */ {

    shared_ptr<FdmLinearOpLayout> layout = mesher->layout();
    vector<Size> newDim(layout->dim());
    iter_swap(newDim.begin(), newDim.begin() + direction_);
    vector<Size> newSpacing = FdmLinearOpLayout(newDim).spacing();
    iter_swap(newSpacing.begin(), newSpacing.begin() + direction_);

    for (FdmLinearOpIterator iter = layout->begin();
          iter != layout->end(); ++iter) {
        const Size i = iter.index();
        i0_[i] = layout->neighbourhood(iter, direction, -1);
        i2_[i] = layout->neighbourhood(iter, direction, 1);
        const vector<Size>& coordinates = iter.coordinates();
        const Size newIndex =
            inner_product(coordinates.begin(), coordinates.end(),
                           newSpacing.begin(), Size(0));
        reverseIndex_[newIndex] = i;
    }
}

Array TripleBandLinearOp::solve_splitting(const Array& r,
                                         Real a, Real b) const {
    // ... initializations ...
    Size rim1 = reverseIndex_[0];
    Real bet=1.0/(a*diag_[rim1]+b);
    retVal[reverseIndex_[0]] = r[rim1]*bet;
    for (Size j=1; j<=layout->size()-1; j++){
        const Size ri = reverseIndex_[j];
        tmp[j] = a*upper_[rim1]*bet;
        bet=1.0/(b+a*(diag_[ri]-tmp[j]*lower_[ri]));
        retVal[ri] = (r[ri]-a*lower_[ri]*retVal[rim1])*bet;
        rim1 = ri;
    }
    for (Size j=layout->size()-2; j>0; --j)
        retVal[reverseIndex_[j]] -=
            tmp[j+1]*retVal[reverseIndex_[j+1]];
    retVal[reverseIndex_[0]] -= tmp[1]*retVal[reverseIndex_[1]];
}

```

---

```
    return retVal;
}
```

---

The basic definition of triple-band operators is similar to that of tridiagonal ones, which you might remember from [the old framework](#). In that case, when applying the operator  $T$  to an array  $x$ , the  $i$ -th element of the result  $y$  could be written as:

$$y_i = l_i x_{i-1} + d_i x_i + u_i x_{i+1}$$

that is, a linear combination of the  $i$ -th element of  $x$  and of its two neighbors ( $l$ ,  $d$  and  $u$  are the lower diagonal, diagonal and upper diagonal). The triple-band operator generalizes the idea to multiple dimensions; here we have, for instance, that the element with indices  $i, j, k, l$  in the result is

$$y_{i,j,k,l} = l_k x_{i,j,k-1,l} + d_k x_{i,j,k,l} + u_k x_{i,j,k+1,l}$$

which is still the combination of the element at the same place in the input and of its two neighbors in a given direction, which is the same for all elements.<sup>19</sup>

When the input and result array are flattened, and thus the operator is written as a 2-dimensional matrix, we find the three bands that give the operator its name. Let's refer again to [the previous figure](#), and assume that the horizontal direction is the first and the vertical direction is the second, and therefore that the elements are enumerated first left to right and then top to bottom. In case (a), the two neighbors are close to the center point in the flattened array, and the operator turns out to be tridiagonal; in case (b), the neighbors are separated by a whole row of elements in the flattened array and the operator has two non-null bands away from the diagonal (which is also not null).

The interface of the class contains a few methods besides the required `apply`. The `solve_splitting` method, like the `solveFor` method in the interface of old-style operators, is used to solve linear systems in which the operator multiplies an array of unknowns, and thus enables to write code using implicit schemes (which are usually stabler or faster than explicit schemes using `apply`); and other methods such as `mult` or `add` define a few operations that make it possible to perform basic algebra and to build operators based on existing pieces (I'll leave it to you to check the code and see what's what).

The constructor performs quite a bit of preliminary computations in order to make the operator ready to use. On the one hand, for each point in the layout, it stores in the two arrays `i0_` and `i2_` the flattened indices of the two neighbors in the given direction; this is done by means of the two calls to `layout->neighborhood`, and allows the core of the `apply` method to be written as a loop similar to:

---

<sup>19</sup>I'm glossing over the elements at the boundaries, but those are easily managed by ignoring their non-existent neighbors.

```
for (Size i=0; i < layout->size(); ++i) {
    result[i] =
        lower_[i]*a[i0_[i]] + diag_[i]*a[i] + upper_[i]*a[i2_[i]];
}
```

where `a` is the array to which the operator is applied, and which translates into code the equation for the triple-band operator.<sup>20</sup>

On the other hand, the constructor fills the `reverseIndex_` array, which provides support for the implementation of `solve_splitting`. In short, and oversimplifying a bit, the algorithm used for solving the system is the same used for tridiagonal operators and involves a double sweep of the array that yields the result in linear time. However, the algorithm is required to sweep the array in the correct order, that is, along the direction of the derivative. Referring again to [the figure above](#), looping over the usual flattened index would sweep the array horizontally, and therefore would only work in case (a) but not in case (b).

The code solves this by introducing a second flattened index. Let's say we have a four-dimensional array with indices  $i, j, k, l$  and corresponding sizes  $M, N, P, Q$ , and that we want to take the derivative along the  $k$  index. The spacing of the layout is  $(1, M, MN, MNP)$ , so the triple  $(i, j, k, l)$  yields the flattened index  $I = i + M \cdot j + MN \cdot k + MNP \cdot l$ . Now, what the constructor does is to create a new spacing with its first dimension swapped with the one corresponding to the derivative; in this example, it would be  $(MN, M, 1, MNP)$ . This defines a new flattened index  $\tilde{I} = MN \cdot i + M \cdot j + k + MNP \cdot l$  that sweeps the array so that consecutive values of  $\tilde{I}$  map to adjacent elements in the  $k$  direction. Finally, the values of  $I$  corresponding to each  $\tilde{I}$  are saved in the `reverseIndex_` array.

This allows the `solve_splitting` method to perform its work correctly. The two loop will enumerate all the values of  $\tilde{I}$  in order, and thus will sweep the array in the correct direction; and the `reverseIndex_` array will give us the corresponding ordinary index that can be used to access the elements into the passed array.

Based on `TripleBandLinearOp`, the framework defines generic operators for the first and second derivative along a given direction; they are sketched in the listing that follows. As you see, they're simple enough: they only declare the constructor, which fills the `lower_`, `diag_` and `upper_` arrays with the correct values (for instance, the coefficients of  $f(x_{i-1})$ ,  $f(x_i)$  and  $f(x_{i+1})$  in the equation for the first derivative) based on the passed layout.

---

<sup>20</sup>The loop in the `apply` method would seem like an obvious candidate for parallelization, and in fact we tried to speed it up by adding a `#pragma omp parallel for`. However, this didn't seem to make the code any faster, and in some cases it actually hurt performance. It might be that the compiler is using parallel instructions already and making a better job when left alone. I'm an absolute beginner at this, though, so I'll leave it to you to investigate the matter if you're interested.

Interface of the `FirstDerivativeOp` and `SecondDerivativeOp` classes.

---

```
class FirstDerivativeOp : public TripleBandLinearOp {
public:
    FirstDerivativeOp(Size direction,
                      const shared_ptr<FdmMesher>& mesher);
};

class SecondDerivativeOp : public TripleBandLinearOp {
public:
    SecondDerivativeOp(Size direction,
                      const shared_ptr<FdmMesher>& mesher);
};
```

---

To round up the basic operators, I'll just mention that the framework also defines a `NinePointLinearOp` class, managing stencils like the one in case (c) of [the previous figure](#), and its derived class `SecondOrderMixedDerivativeOp`, modeling the cross-derivative  $\frac{\partial^2 f}{\partial x_i \partial x_j}$  along two directions.

### 8.2.3 Examples: Black-Scholes operators

Most full-featured operators in the library are not inherited directly from the `FdmLinearOp` class or from one of the basic operators. Instead, they inherit from the `FdLinearOpComposite` class (shown in the next listing), contain one or more basic operators as data members, and use them to implement their own behavior.

Interface of the `FdLinearOpComposite` class.

---

```
class FdLinearOpComposite : public FdmLinearOp {
public:
    virtual Size size() const = 0;

    virtual void setTime(Time t1, Time t2) = 0;

    virtual Array apply_mixed(const Array& r) const = 0;
    virtual Array apply_direction(Size direction,
                                  const Array& r) const = 0;
    virtual Array solve_splitting(Size direction,
                                 const Array& r,
                                 Real s) const = 0;
    virtual Array preconditioner(const Array& r,
                                Real s) const = 0;
};

};
```

---

As you can see, the `FdmLinearOpComposite` class augments the interface of operators with a few more methods that derived classes must implement, the simplest being `size` (which must return the number of dimensions of the operator).

The other methods deserve a bit more attention. The `setTime` method implements the same idea I described in the section on time-dependent operators: when called, it should modify or rebuild a time-dependent operator so that its elements correspond to the correct time. However, there are a couple of enhancements. First, in the old framework the machinery was added to the `TridiagonalOperator` class, that is, to the basic building block for operators; in the new one, basic operators are constant (and simpler) and the `setTime` method was added to a higher-level interface. Second, the operator will be used over a given step, and this interface allows one to pass both its start and end time; this allows operators to provide a better discretization (for instance, by averaging or integrating) than simply the value at a single time.

The remaining methods are used to support Alternating Direction Implicit (ADI) schemes. I'm not going to describe them in any detail here;<sup>21</sup> the basic idea is that an operator  $A$  is decomposed as  $A = A_m + A_0 + A_1 + \dots + A_{N-1}$ , where  $A_m$  represents the mixed derivatives and each  $A_i$  represents the derivatives along the  $i$ -th direction, and those components are used separately. Instead of having the operator create the actual components and let the schemes apply them, the interface declares corresponding methods: thus, the `apply_mixed` method will apply the  $A_m$  component and the `apply_direction` method will apply one of the  $A_i$  depending on the direction passed to the method. The `solve_splitting` and `preconditioner` methods are also used in ADI schemes.

As a first example, you can look at the `FdmBlackScholesOp` class, shown in the following listing.

Sketch of the `FdmBlackScholesOp` class.

---

```
class FdmBlackScholesOp : public FdmLinearOpComposite {
public:
    FdmBlackScholesOp(
        const shared_ptr<FdmMesher>& mesher,
        const shared_ptr<GeneralizedBlackScholesProcess>&,
        Real strike,
        bool localVol = false,
        Real illegalLocalVolOverwrite = -Null<Real>(),
        Size direction = 0);

    Size size() const { return 1; }
    void setTime(Time t1, Time t2) {
        if (localVol_) {
            /* `more calculations` */
        } else {
            Rate r = rTS_->forwardRate(t1, t2, Continuous);
            Rate q = qTS_->forwardRate(t1, t2, Continuous);
    }
}
```

---

<sup>21</sup>A summary is available, e.g., in [\(de Graaf, 2012\)](#), and you all know how to use Google anyway.

```

        Real v = volTS_->blackForwardVariance(
            t1, t2, strike_)/(t2-t1);
        mapT_ = /* `put them together` */;
    }
}

Array apply(const Array& r) const {
    return mapT_.apply(r);
}
Array apply_mixed(const Array& r) const {
    return Array(r.size(), 0.0);
}
Array apply_direction(Size direction,
                      const Array& r) const {
    return (direction == direction_) ? mapT_.apply(r)
                                    : Array(r.size(), 0.0);
}
Array solve_splitting(Size direction,
                      const Array& r, Real s) const;
Array preconditioner(const Array& r, Real s) const;
private:
    TripleBandLinearOp mapT_;
    /* ... other data members ... */
};
```

---

As expected, it inherits from `FdmLinearOpComposite` and implements its required interface. The constructor takes quite a few parameters and stores them in the corresponding data members for later use; among them, it takes a given direction, which let us specify the axis along which this operator works and thus allows it to be used as a building block in other operators.

The underlying differential operator, stored in the `mapT_` data member, is built inside the `setTime` method. As I mentioned, having both the start time `t1` and the end time `t2` of the step allows the code to provide a more accurate discretization; namely, it can ask the term structures for the exact forward rates and variance between those two times instead of picking an instantaneous value. Depending on whether we want to use local volatility, the calculation of the diffusion term differs; but in both cases we end up building the Black-Scholes operator

$$L_{BS}(t) = - \left( r - q - \frac{\sigma^2}{2} \right) \frac{\partial}{\partial x} - \frac{\sigma^2}{2} \frac{\partial^2}{\partial x^2} + r$$

and storing it into `mapT_` so that it can be used elsewhere.

Once the final operator is built, the implementation of the remaining methods is straightforward enough. The `apply` method applies `mapT_` to the passed array and return the results. Since this is

a one-dimensional operator, there are no cross-terms, therefore `apply_mixed` returns a null array. Finally, the `apply_direction` method applies `mapT_` to the passed array when the direction equals the one specified in the constructor (because that's the one direction along which the entire operator works) and instead returns a null array when the direction is different (because the operator has no corresponding component). The implementations of the other methods work in a similar way.

As a second example, the `Fdm2dBlackScholesOp` class, shown in the next listing, builds a two-dimensional Black-Scholes operator by combining a pair of one-dimensional operators with their correlation.

Sketch of the `Fdm2dBlackScholesOp` class.

---

```

class Fdm2dBlackScholesOp : public FdmLinearOpComposite {
    public:
        Fdm2dBlackScholesOp(
            const shared_ptr<FdmMesher>& mesher,
            const shared_ptr<GeneralizedBlackScholesProcess>& p1,
            const shared_ptr<GeneralizedBlackScholesProcess>& p2,
            Real correlation,
            Time maturity,
            bool localVol = false,
            Real illegalLocalVolOverwrite = -Null<Real>());
        Size size() const { return 2; }
        void setTime(Time t1, Time t2) {
            opX_.setTime(t1, t2);
            opY_.setTime(t1, t2);

            corrMapT_ = /* ... other calculations ... */
        }

        Array apply(const Array& x) const {
            return opX_.apply(x) + opY_.apply(x) + apply_mixed(x);
        }
        Array apply_mixed(const Array& x) const {
            return corrMapT_.apply(x) + currentForwardRate_*x;
        }
        Array apply_direction(Size direction,const Array& x) const {
            if (direction == 0)
                return opX_.apply(x);
            else if (direction == 1)
                return opY_.apply(x);
            else
                QL_FAIL("direction is too large");
        }
}
```

---

```

        Array solve_splitting(Size direction,
                           const Array& x, Real s) const;
    Array preconditioner(const Array& r, Real s) const;
private:
    FdmBlackScholesOp opX_, opY_;
    NinePointLinearOp corrMapT_;
/* ... other data members ... */
};
```

---

Again, the constructor stores the passed data, while the `setTime` method builds the operators; directly in the case of the correlation `corrMapT_`, and forwarding to the corresponding methods in the case of the two one-dimensional operators `opX_` and `opY_`.

The other methods use the previously built operators. Given their linearity, the `apply` method works by applying each component to the passed array in turn and returning the sum of the results; `apply_mixed` uses the mixed operator `corrMapT_` plus a constant term; and `apply_direction` selects and applies the correct one-dimensional component.

If you're interested, the library provides other operators you can examine. Most of them turn out to have the same structure: for instance, the `FdmHestonOp` class, which implements a helper operator for each underlying variable and puts them together in the final operator, or the `FdmHestonHullWhiteOp`, that you can inspect as an example of three-dimensional operator.

## 8.2.4 Initial, boundary, and step conditions

In the old framework, as you might remember, base engines declared a virtual method that derived classes had to implement and which applied the initial condition of the problem. In the new framework, in keeping with its plug-and-play nature, calculating the initial condition is left instead to separate objects that can be reused in different engines.

The base class for such objects is `FdmInnerValueCalculator`, shown in the following listing.

Interface of the `FdmInnerValueCalculator` class and implementation of a few derived classes.

---

```

class FdmInnerValueCalculator {
    public:
        virtual ~FdmInnerValueCalculator() {}

        virtual Real innerValue(const FdmLinearOpIterator& iter,
                           Time t) = 0;
        virtual Real avgInnerValue(const FdmLinearOpIterator& iter,
                           Time t) = 0;
};

class FdmLogInnerValue : public FdmInnerValueCalculator {
```

```

public:
    FdmLogInnerValue(const shared_ptr<Payoff>& payoff,
                      const shared_ptr<FdmMesher>& mesher,
                      Size direction);

    Real innerValue(const FdmLinearOpIterator& iter, Time) {
        Real s = std::exp(mesher_->location(iter, direction_));
        return (*payoff_)(s);
    }
    Real avgInnerValue(const FdmLinearOpIterator& iter, Time);
};

class FdmLogBasketInnerValue : public FdmInnerValueCalculator {
public:
    FdmLogBasketInnerValue(
        const shared_ptr<BasketPayoff>& payoff,
        const shared_ptr<FdmMesher>& mesher);

    Real innerValue(const FdmLinearOpIterator& iter, Time) {
        Array x(mesher_->layout()->dim().size());
        for (Size i=0; i < x.size(); ++i)
            x[i] = std::exp(mesher_->location(iter, i));
        return (*payoff_)(x);
    }
    Real avgInnerValue(const FdmLinearOpIterator& iter, Time) {
        return innerValue(iter, t);
    }
};

```

---

Its interface contains the `innerValue` method, which returns the initial value at a given point in the mesh (denoted by an iterator) and the `avgInnerValue` method, which returns an average of the initial value around a given point and thus might provide a way to smooth discontinuities in the payoff. They're both declared as pure virtual; this makes complete sense for `innerValue`, and is not wrong for `avgInnerValue` either. In hindsight, though, a few derived classes give up the calculation of the average as too difficult and return the value at the central point (i.e., the return value of `innerValue`) as an approximation; this might have been a default implementation.<sup>22</sup>

The listing also contains a couple of examples of concrete classes implementing the interface. The `FdmLogInnerValue` class is used to calculate a payoff, depending on a single underlying variable, on a logarithmic mesh. It takes the payoff itself, the mesh, and the index of the axis along which

---

<sup>22</sup>You could also argue that the approximation should be a deliberate choice of whoever writes the derived class, and you wouldn't be wrong either.

the underlying varies; the latter is needed because the mesh might be multi-dimensional.<sup>23</sup> As you probably expect, the `innerValue` method extracts from the mesh the value of the underlying variable corresponding to the passed iterator, and then passes it to the payoff. The `avgInnerValue` method does something similar, but with a combination of numerical integrals and `boost::bind` that I won't bother to show here (and that I hope to eradicate when we can use lambdas instead).

The second example, `FdmLogBasketInnerValue`, is used for a payoff depending on the values of multiple underlying variables (and not on the total basket value, which probably makes the class name a misnomer). Its `innerValue` method collects the underlying values from the several axis of the mesh—which, in this case, is assumed not to model any other variable—and passes them to the payoff. The `avgInnerValue` method bails out (poor thing) and calls `innerValue`.

### Aside: dispelling magic.

If you look back at the creation of the inner-value calculator in the engine code shown at the beginning of the section, you'll see that we pass the index of the dimension as the magic number 0. This, of course, is not ideal. The problem is that the info is hard-coded in the definition of the mesher, but it's not accessible by code, and it's not easy to think of a meaningful common interface that meshers might provide to retrieve it.

I don't know if there's a solution, other than the time-honored mitigation method of giving magic numbers a name; for instance, the `FdmBlackScholesMesher` class could provide a specific static variable such as `x_dimension`. However, good names for these variables might be hard to define; and the idea runs counter to a possible simplification I mentioned earlier, in which meshers could be non-polymorphic and derived classes would be replaced by factory functions.

The new framework doesn't declare any new interface for boundary conditions; the `FdmBoundaryConditionSet` seen in the engine is defined as

```
typedef OperatorTraits<FdmLinearOp>::bc_set  
FdmBoundaryConditionSet;
```

where `OperatorTraits` comes from the old framework and thus makes use of the `BoundaryCondition` class template I described in [a previous section](#); the only thing new is that we're instantiating it with the `FdmLinearOp` class.

Of course, specific boundary conditions are implemented in this framework so that they work with the new classes. For instance, the next listing shows the interface of the `FdmDirichletBoundary` class. Its constructor takes the mesher that defines the domain of the problem, the value of the instrument on the boundary, the index of the axis whose boundary we want to control and an enumeration to specify the lower or upper side of the mesh. The implementation, not shown, also uses the mesher to identify the array elements to condition.

---

<sup>23</sup>For instance, the underlying might be a stock value and the problem might model the stock and its volatility, or the stock and the interest rate, or all three. There's no guarantee that the stock varies along the first axis.

Partial interface of the `FdmDirichletBoundary` class.

---

```
class FdmDirichletBoundary
    : public BoundaryCondition<FdmLinearOp> {
public:
    FdmDirichletBoundary(const shared_ptr<FdmMesher>& mesher,
                         Real valueOnBoundary,
                         Size direction, Side side);
    void applyBeforeApplying(operator_type&) const;
    void applyBeforeSolving(operator_type&, array_type&) const;
    void applyAfterApplying(array_type&) const;
    void applyAfterSolving(array_type&) const;
    void setTime(Time) {}
};
```

---

Finally, step conditions also inherit from a class template defined in the old framework (`StepCondition`, also described [before](#)); and of course, they also use the classes defined in the new framework. One such condition is implemented in the `FdmAmericanStepCondition` class, shown in the listing that follows, which determines if an American option should be exercised.

Implementation of the `FdmAmericanStepCondition` class.

---

```
class FdmAmericanStepCondition : public StepCondition<Array> {
public:
    FdmAmericanStepCondition(
        const shared_ptr<FdmMesher>&,
        const shared_ptr<FdmInnerValueCalculator>&);

    void applyTo(Array& a, Time) const {
        shared_ptr<FdmLinearOpLayout> layout = mesher_->layout();

        for (FdmLinearOpIterator iter = layout->begin();
             iter != layout->end(); ++iter) {
            Real innerValue = calculator_->innerValue(iter, t);
            if (innerValue > a[iter.index()]) {
                a[iter.index()] = innerValue;
            }
        }
    }

private:
    shared_ptr<FdmMesher> mesher_;
    shared_ptr<FdmInnerValueCalculator> calculator_;
};
```

---

The implementation is how you would expect: the `applyTo` method iterates over the mesh, and at each point it compares the intrinsic value given by the option payoff with the current value of the option and it updates the corresponding array element. The constructor takes and stores the information that `applyTo` needs.

The framework also defines the `FdmStepConditionComposite` class. It is a convenience class, which uses the Composite pattern to group multiple step conditions and pass them around as a single entity. Its interface is shown in the next listing; it defines the required `applyTo` method, which applies all conditions in sequence, as well as a number of utility methods. Most of them are generic enough; there are inspectors for the stored information, as well as a method to join multiple conditions (or groups thereof).

Interface of the `FdmStepConditionComposite` class.

---

```

class FdmStepConditionComposite : public StepCondition<Array> {
    public:
        typedef list<shared_ptr<StepCondition<Array>> > Conditions;

        FdmStepConditionComposite(
            const list<vector<Time>> & stoppingTimes,
            const Conditions & conditions);

        void applyTo(Array& a, Time t) const;

        const vector<Time>& stoppingTimes() const;
        const Conditions& conditions() const;

        static shared_ptr<FdmStepConditionComposite> joinConditions(
            const shared_ptr<FdmSnapshotCondition>& c1,
            const shared_ptr<FdmStepConditionComposite>& c2);

        static shared_ptr<FdmStepConditionComposite>
        vanillaComposite(
            const DividendSchedule& schedule,
            const shared_ptr<Exercise>& exercise,
            const shared_ptr<FdmMesher>& mesher,
            const shared_ptr<FdmInnerValueCalculator>& calculator,
            const Date& refDate,
            const DayCounter& dayCounter);
    };

```

---

However, the `vanillaComposite` static method is a bit different. It is a factory method that takes a bunch of arguments and makes a ready-to-go composite condition that can be used in different problems; which is good, but it's more specific than the rest of the class. Not that specific, mind

you, since a quick search shows that it's used in engines for a few instruments as different as barrier options and swaptions; but still, if we were to write more of them, I wouldn't like them to proliferate inside this class. It might be cleaner to define them as free functions.

### 8.2.5 Schemes and solvers

As in the old framework, schemes provide a discretization of the time derivative in the equation we want to solve; you can go [back to the old framework](#) for more details. The new framework provides reimplementations of schemes used in the old one, such as explicit and implicit Euler, as well as a number of new ones I won't describe; as usual, you can look them up in the library.

Like in the old framework, they have no common base class; they all courteously agree to implement `step` and `setStep` methods with the same semantics described in the section I just quoted, but there's no inheritance relationship holding them to any contract. This is a bit of a problem, given that the new framework usually eschews generic programming and templates in favor of more object-oriented techniques; the absence of a base class, of course, prevents us from writing non-template code taking an instance of a generic scheme.

The way the framework works around this is to define the `FdmSchemeDesc` class, shown in the following listing. Its instances can be passed around instead of the scheme they describe, and contain the type of the chosen scheme and the parameters that might be needed to instantiate it. The available types are enumerated in the class declaration, and yes, that's not optimal. I'll get back to this later.

Interface of the `FdmSchemeDesc` class.

---

```
struct FdmSchemeDesc {
    enum FdmSchemeType { HundsdorferType, DouglasType,
                          CraigSneydType, ModifiedCraigSneydType,
                          ImplicitEulerType, ExplicitEulerType };

    FdmSchemeDesc(FdmSchemeType type, Real theta, Real mu);

    const FdmSchemeType type;
    const Real theta, mu;

    static FdmSchemeDesc Douglas();
    static FdmSchemeDesc ImplicitEuler();
    static FdmSchemeDesc ExplicitEuler();
    static FdmSchemeDesc CraigSneyd();
    static FdmSchemeDesc ModifiedCraigSneyd();
    static FdmSchemeDesc Hundsdorfer();
    static FdmSchemeDesc ModifiedHundsdorfer();
};
```

---

Besides the enumeration, the class declares a bunch of static methods returning a few pre-built schemes with sensible parameters; if you want a custom one instead, you can build an instance with any old values.

Note also that this class and the others described in this section are not templates; they're written to work specifically with the `FdmLinearOpComposite` class. My guess is that their authors learned the lesson from the old framework, which was written for a generic operator and turned out to be only ever used with the `TridiagonalOperator` class.

Now, if the type enumeration seemed a bit icky to you, you might want to brace yourself for the next listing and the `FdmBackwardSolver` class.

Sketch of the **FdmBackwardSolver** class.

---

```

        map_, bcSet_);
FiniteDifferenceModel<HundsdoerferScheme>
hsModel(hsEvolver, condition_->stoppingTimes());
hsModel.rollback(rhs, dampingTo, to,
                 steps, *condition_);
}
break;
// other cases, not shown
}
}
};
```

---

Its constructor collects and stores an operator, a set of boundary and step conditions, and a scheme description. They are used in the `rollback` method, which puts them together in a finite-difference model, takes an array of initial values, and runs the model between the two given times `from` and `to` in the given number of steps, possibly with a few initial damping steps. And, as you can see, the implementation of `rollback` is a big switch on type.

Was that a gasp? Yes, I know. Both the `FdmSchemeDesc` class and the `rollback` method need to be changed if we want to add a new scheme, which is a violation of the open-closed principle. It's not like the authors of the code were sloppy, though. Let's say we go ahead and inherit all schemes from a base class, so we can use a polymorphic pattern such as Strategy to select one. There are two forces at play here. First, if the user must be able to choose it, the scheme instance would have to be passed to the engine from outside. Second, the operator and the boundary conditions, which the scheme requires, are created inside the engine, because having the user write code to create them would cause duplication, and thus the scheme must be created inside the engine, too. Tricky, isn't it?

Two possible solutions come to mind. One is to use some kind of factory: the scheme might have, say, a `clone` method that returns another instance with the same type and parameters and with the passed boundary and step conditions added.<sup>24</sup> Another is to change the interface of the classes we've seen, so that the scheme doesn't contain the conditions; since backward compatibility prevents us from changing the `FiniteDifferenceModel` class, this requires writing a new model class, too. Both would have been possible, and the maintainer who reviewed and merged the framework when it was contributed should have seen the problem and should have suggested one of them. That maintainer was me, in case you're curious. Live and learn, I guess.

And after the `FdmBackwardSolver` class, we're on the home stretch; the remaining classes add a couple of layers that make it more convenient to use the solver, but not a lot of new logic. To the more observant among you: yes, in this section I abandoned my initial intent to write a top-down description of the components. I think it makes more sense this way.

First, the `FdmSolverDesc` structure shown in the listing that follows groups together the arguments needed to initialize a solver so that we can pass them around together. The operator is not included;

---

<sup>24</sup>We could also have factory classes, but that would mean a parallel hierarchy, which is never good.

that's because, as you'll see in a minute, we'll define higher-level solvers that define the operator and can be reused for different instruments by passing them different `FdmSolverDesc` instances.

Interface of the `FdmSolverDesc` struct.

---

```
struct FdmSolverDesc {
    const shared_ptr<FdmMesher> mesher;
    const FdmBoundaryConditionSet bcSet;
    const shared_ptr<FdmStepConditionComposite> condition;
    const shared_ptr<FdmInnerValueCalculator> calculator;
    const Time maturity;
    const Size timeSteps;
    const Size dampingSteps;
};
```

---

Second, the `Fdm1DimSolver` class sketched in the next listing takes care of some of the plumbing around a solver.

Partial implementation of the `Fdm1DimSolver` class.

---

```
class Fdm1DimSolver : public LazyObject {
public:
    Fdm1DimSolver(const FdmSolverDesc& solverDesc,
                  const FdmSchemeDesc& schemeDesc,
                  const shared_ptr<FdmLinearOpComposite>& op)
    : /* ... */ {
        // ... get mesher, calculator etc. from solver description ...
        FdmLinearOpIterator end = layout->end();
        for (FdmLinearOpIterator i = layout->begin();
              i != end; ++i) {
            initialValues_[i.index()] =
                calculator->avgInnerValue(i, maturity);
            x_[i.index()] = mesher->location(i, 0);
        }
    }

    Real interpolateAt(Real x) const {
        calculate();
        return (*interpolation_)(x);
    }
    // ...other inspectors for derivatives
protected:
    void performCalculations() const {
        Array rhs(initialValues_.size());
```

```

    std::copy(initialValues_.begin(), initialValues_.end(),
              rhs.begin());

    FdmBackwardSolver(op_, solverDesc_.bcSet,
                      conditions_, schemeDesc_)
        .rollback(rhs, solverDesc_.maturity, 0.0,
                  solverDesc_.timeSteps,
                  solverDesc_.dampingSteps);

    std::copy(rhs.begin(), rhs.end(), resultValues_.begin());
    interpolation_ = shared_ptr<CubicInterpolation>(
        new MonotonicCubicNaturalSpline(
            x_.begin(), x_.end(), resultValues_.begin()));
}
};


```

---

Its constructor takes a solver description, a scheme description and an operator, stores them, and prepares a number of other data members for the calculations. In particular, it uses the layout in the solver description to allocate the correct size for the initial values, and the calculator to fill them; also, it extracts from the mesher the values of the underlying variable on the chosen grid.

The calculations are implemented inside the `performCalculations` method (yes, the class inherits from `LazyObject`, which puzzles me; I'll come back to this). As I mentioned, it's mostly plumbing: the code makes a copy of the initial values, creates an instance of `FdmBackwardSolver`, uses it to roll back the values, stores the results, and wraps them in an interpolation. Methods such as `interpolateAt` make the results available to client code; other inspectors, not shown, give access to the first and second derivative of the interpolation, as well as the time derivative of the result.

Going back one step: I gave the matter some thought, but I don't see the need for inheriting from `LazyObject` here. For one thing, the constructor of `Fdm1DimSolver` doesn't register with anything, which kind of defeats the purpose of the pattern.<sup>25</sup> Moreover, we'll see that solvers are normally created by an engine during calculation, used right there, and discarded afterwards. All in all, we could have put the whole calculation in the constructor, or even turned the class into a function returning some kind of result structure containing the interpolation and some inspectors to access it.

I should mention that `Fdm1DimSolver` has siblings: the framework also defines the `Fdm2DimSolver` and `Fdm3DimSolver` classes, which perform the same calculations and produce a 2-D or 3-D interpolation, respectively. The interface is almost the same, with some difference due to the higher dimensions; for instance, the `interpolateAt` method takes as arguments `x` and `y` in the 2-D case and `x`, `y` and `z` in the 3-D case.

Finally, specific solvers are tasked with creating an operator and using it with one of the generic solvers I described. It's the case of the `FdmBlackScholesSolver` class, shown in the next listing.

<sup>25</sup>Registration might happen in derived classes, but a quick search in the library shows that `Fdm1DimSolver` doesn't have any.

Partial implementation of the `FdmBlackScholesSolver` class.

---

```

class FdmBlackScholesSolver : public LazyObject {
    public:
        FdmBlackScholesSolver(
            const Handle<GeneralizedBlackScholesProcess>& process,
            Real strike,
            const FdmSolverDesc& solverDesc,
            const FdmSchemeDesc& schemeDesc,
            bool localVol = false,
            Real illegalLocalVolOverwrite = -Null<Real>())
        : /* ... */ {
            registerWith(process_);
        }

        Real valueAt(Real s) const {
            calculate();
            return solver_->interpolateAt(std::log(s));
        }
        Real deltaAt(Real s) const {
            calculate();
            return solver_->derivativeX(std::log(s))/s;
        }
        Real gammaAt(Real s) const;
        Real thetaAt(Real s) const;

    protected:
        void performCalculations() const {
            const shared_ptr<FdmBlackScholesOp> op(
                new FdmBlackScholesOp(
                    solverDesc_.mesh, process_.currentLink(),
                    strike_, localVol_, illegalLocalVolOverwrite_));

            solver_ = shared_ptr<Fdm1DimSolver>(
                new Fdm1DimSolver(solverDesc_, schemeDesc_, op));
        }

    private:
        // other data members, not shown
        mutable shared_ptr<Fdm1DimSolver> solver_;
};
```

---

Besides the usual descriptions and a few other parameters for the model, it takes a handle to a Black-

Scholes process that it stores and with which it registers for notifications; again, this would make more sense if engines kept the solver around instead of destroying it. Its `performCalculations` method creates an operator based on the process and uses it to build a 1-D solver, which is then stored. Inspectors such as `valueAt` and `deltaAt` call `calculate` first, which triggers the construction of the 1-D solver, and then delegate to the corresponding methods of the stored solver, which in turn trigger the rollback calculation inside it. The inspectors also take care of the transformation between the underlying value, used in the interface, and its logarithm, used in the process.

As I mentioned earlier, the solver takes the solver description from the outside and thus can be used for different engines, provided that the underlying of the option follows the same dynamics. In the listing at [the beginning of the section](#), you can see it used for a vanilla put or call option; by changing the boundary conditions, it can be used, and in fact it is used, for barrier options; and by changing the initial conditions, it might be used for digital options or other kinds of instruments. You can see how this design is more flexible than the old, inheritance-based one; or at least, that it is much simpler to get flexibility.

Multi-dimensional solvers, which I won't describe, work in the same way. If you're interested, you can look at the `FdmBatesSolver` class for a 2-D example and at the `FdmHestonHullWhiteSolver` class for a 3-D one. And, as the saying goes, there's plenty more where those come from.

\* \* \*

With solvers, we finally have all the pieces we need to build our engines—and round up this chapter, I might add. Looking back at the listing of [the engine we took as example](#), we can now see how a finite-difference engine works. It takes the scheme and the other required parameters at construction, and then performs its calculations by building the required mesher, the initial, step and boundary conditions, and finally the solver from which the results can be extracted.

# 9. Conclusion

Thank you for reading so far. There are a few *encores* for you in the appendix, but the main content is over and so it's time for a few parting thoughts from yours truly.

Writing a book is no easy feat. The first time, if one's not completely unaware (which I might have been, seeing that I started), it's likely to cause the feelings expressed by the bogus Wizard of Oz in the 1939 movie:

Frightened? Child, you're talking to a man who's laughed in the face of death, sneered at doom, and chuckled at catastrophe... I was petrified.

Now, if you've been following me for a while, you know I had a pretty long time to get comfortable with the idea. But while I'm no longer petrified, I suppose it's only natural that I've been asking myself whether I should have done a few things in a different way.<sup>1</sup> Also, getting to the end does put one into a reflective mood of sorts—especially after a pint or two.

In particular, looking back at the previous chapters, I see that sometimes I've been the worst critic of our library and I focused more on what I would change if I were to write it anew. (True story: someone actually told me that he decided not to use QuantLib in a simulation because of what I had written. That's how good at marketing I am.) So, let me say it out loud: when writing about the code, I always tried to remember that the library was written by capable, well-intentioned people that worked within the constraints they had at the time. In this specific case, this was easier for me to do because those people would often be past versions of me and of friends of mine; but it will do us good to apply this to programming at large, if not to the whole world outside.

With this in mind, I'll keep those critiques as they are. As I wrote in the introduction, I wrote this book primarily for QuantLib users but I hope you can find it useful even if you decide that QuantLib is not for you. I've tried to give you the reasons for what we did, and the different ways I would do it now with other constraints, so that you can make more informed choices in your own coding. *Il che è bello e istruttivo*, as Giovannino Guareschi used to say.

Let me know if it worked.

---

<sup>1</sup>For instance, I didn't manage to work in a *Firefly* reference, even though I aimed to misbehave.

# A. Odds and ends

A number of basic issues with the usage of QuantLib have been glossed over in the previous chapters, in order not to undermine their readability (if any) with an accumulation of technical details; as pointed out by Douglas Adams in the fourth book of its *Hitchhiker* trilogy.<sup>2</sup>

[An excessive amount of detail] is guff. It doesn't advance the action. It makes for nice fat books such as the American market thrives on, but it doesn't actually get you anywhere.

This appendix provides a quick reference to some such issues. It is not meant to be exhaustive nor systematic;<sup>3</sup> if you need that kind of documentation, check the QuantLib reference manual, available at the [QuantLib web site](#).

## Basic types

The library interfaces don't use built-in types; instead, a number of typedefs are provided such as `Time`, `Rate`, `Integer`, or `Size`. They are all mapped to basic types (we talked about using full-featured types, possibly with range checking, but we dumped the idea). Furthermore, all floating-point types are defined as `Real`, which in turn is defined as `double`. This makes it possible to change all of them consistently by just changing `Real`.

In principle, this would allow one to choose the desired level of accuracy; but to this, the test-suite answers "Fiddlesticks!" since it shows a few failures when `Real` is defined as `float` or `long double`. The value of the typedefs is really in making the code more clear—and in allowing dimensional analysis for those who, like me, were used to it in a previous life as a physicist; for instance, expressions such as `exp(r)` or `r+s*t` can be immediately flagged as fishy if they are preceded by `Rate r`, `Spread s`, and `Time t`.

Of course, all those fancy types are only aliases to `double` and the compiler doesn't really distinguish between them. It would be nice if they had stronger typing; so that, for instance, one could overload a method based on whether it is passed a price or a volatility.

One possibility would be the `BOOST_STRONG_TYPEDEF` macro, which is one of the bazillion utilities provided by Boost. It is used as, say,

---

<sup>2</sup>No, it's not a mistake. It is an inaccurately-named trilogy of five books. It's a long story.

<sup>3</sup>Nor automatic, nor hydromatic. That would be the Grease Lightning.

```
BOOST_STRONG_TYPEDEF(double, Time)
BOOST_STRONG_TYPEDEF(double, Rate)
```

and creates a corresponding proper class with appropriate conversions to and from the underlying type. This would allow overloading methods, but has the drawbacks that not all conversions are explicit. This would break backward compatibility and make things generally awkward.<sup>4</sup>

Also, the classes defined by the macro overload all operators: you can happily add a time to a rate, even though it doesn't make sense (yes, dimensional analysis again). It would be nice if the type system prevented this from compiling, while still allowing, for instance, to add a spread to a rate yielding another rate or to multiply a rate by a time yielding a pure number.

How to do this in a generic way, and ideally with no run-time costs, was shown first by [Barton and Nackman, 1995](#); a variation of their idea is implemented in the Boost.Units library, and a simpler one was implemented once by yours truly while still working in Physics.<sup>5</sup> However, that might be overkill here; we don't have to deal with all possible combinations of length, mass, time and so on.

The ideal compromise for a future library might be to implement wrapper classes (à la Boost strong typedef) and to define explicitly which operators are allowed for which types. As usual, we're not the first ones to have this problem: the idea has been floating around for a while, and at some point a proposal was put forward ([Brown, 2013](#)) to add to C++ a new feature, called *opaque typedefs*, which would have made it easier to define this kind of types.

A final note: among these types, there is at least one which is not determined on its own (like `Rate` or `Time`) but depends on other types. The volatility of a price and the volatility of a rate have different dimensions, and thus should have different types. In short, `Volatility` should be a template type.

## Date calculations

Date calculations are among the basic tools of quantitative finance. As can be expected, QuantLib provides a number of facilities for this task; I briefly describe some of them in the following subsections.

### Dates and periods

An instance of the `Date` class represents a specific day such as November 15th, 2014. This class provides a number of methods for retrieving basic information such as the weekday, the day of the month, or the year; static information such as the minimum and maximum date allowed (at this time, January 1st, 1901 and December 31st, 2199, respectively) or whether or not a given year is a leap year; or other information such as a date's Excel-compatible serial number or whether or not a given date is the last date of the month. The complete list of available methods and their interface

---

<sup>4</sup>For instance, a simple expression like `Time t = 2.0;` wouldn't compile. You'd also have to write `f(Time(1.5))` instead of just `f(1.5)`, even if `f` wasn't overloaded.

<sup>5</sup>I won't explain it here, but go read it. It's almost insanely cool.

is documented in the reference manual. No time information is included (unless you enable an experimental compilation switch).

Capitalizing on C++ features, the `Date` class also overloads a number of operators so that date algebra can be written in a natural way; for example, one can write expressions such as `++d`, which advances the date `d` by one day; `d + 2`, which yields the date two days after the given date; `d2 - d1`, which yields the number of days between the two dates; `d - 3*Weeks`, which yields the date three weeks before the given date (and incidentally, features a member of the available `TimeUnit` enumeration, the other members being `Days`, `Months`, and `Years`); or `d1 < d2`, which yields `true` if the first date is earlier than the second one. The algebra implemented in the `Date` class works on calendar days; neither bank holidays nor business-day conventions are taken into account.

The `Period` class models lengths of time such as two days, three weeks, or five years by storing a `TimeUnit` and an integer. It provides a limited algebra and a partial ordering. For the non mathematically inclined, this means that two `Period` instances might or might not be compared to see which is the shorter; while it is clear that, say, 11 months are less than one year, it is not possible to determine whether 60 days are more or less than two months without knowing *which* two months. When the comparison cannot be decided, an exception is thrown.

And of course, even when the comparison seems obvious, we managed to sneak in a few surprises. For instance, the comparison

```
Period(7,Days) == Period(1,Weeks)
```

returns `true`. It seems correct, right? Hold that thought.

## Calendars

Holidays and business days are the domain of the `Calendar` class. Several derived classes exist which define holidays for a number of markets; the base class defines simple methods for determining whether or not a date corresponds to a holiday or a business day, as well as more complex ones for performing tasks such as adjusting a holiday to the nearest business day (where “nearest” can be defined according to a number of business-day conventions, listed in the `BusinessDayConvention` enumeration) or advancing a date by a given period or number of business days.

It might be interesting to see how the behavior of a calendar changes depending on the market it describes. One way would have been to store in the `Calendar` instance the list of holidays for the corresponding market; however, for maintainability we wanted to code the actual calendar rules (such as “the fourth Thursday in November” or “December 25th of every year”) rather than enumerating the resulting dates for a couple of centuries. Another obvious way would have been to use polymorphism and the Template Method pattern; derived calendars would override the `isBusinessDay` method, from which all others could be implemented. This is fine, but it has the shortcoming that calendars would need to be passed and stored in `shared_ptrs`. The class is conceptually simple, though, and is used frequently enough that we wanted users to instantiate it and pass it around more easily—that is, without the added verbosity of dynamic allocation.

The final solution was the one shown in the listing below. It is a variation of the *pimpl* idiom, also reminiscent of the Strategy or Bridge patterns; these days, the cool kids might call it *type erasure*, too (Becker,2007).

Outline of the `Calendar` class.

---

```

class Calendar {
    protected:
        class Impl {
            public:
                virtual ~Impl() {}
                virtual bool isBusinessDay(const Date&) const = 0;
            };
            shared_ptr<Impl> impl_;
        public:
            bool isBusinessDay(const Date& d) const {
                return impl_->isBusinessDay(d);
            }
            bool isHoliday(const Date& d) const {
                return !isBusinessDay(d);
            }
            Date adjust(const Date& d,
                        BusinessDayConvention c = Following) const {
                // uses isBusinessDay() plus some logic
            }
            Date advance(const Date& d,
                         const Period& period,
                         BusinessDayConvention c = Following,
                         bool endOfMonth = false) const {
                // uses isBusinessDay() and possibly adjust()
            }
            // more methods
        };

```

---

Long story short: `Calendar` declares a polymorphic inner class `Impl` to which the implementation of the business-day rules is delegated and stores a pointer to one of its instances. The non-virtual `isBusinessDay` method of the `Calendar` class forwards to the corresponding method in `Calendar::Impl`; following somewhat the Template Method pattern, the other `Calendar` methods are also non-virtual and implemented (directly or indirectly) in terms of `isBusinessDay`.<sup>6</sup>

Coming back to this after all these years, though, I'm thinking that we might have implemented all public methods in terms of `isHoliday` instead. Why? Because all calendars are defined by stating which days are holidays (e.g., Christmas on December 25th in a lot of places, or MLK Day on the

---

<sup>6</sup>The same technique is used in a number of other classes, such as `DayCounter` in the next section or `Parameter` from chapter 5.

third Monday in January in the United States). Having `isBusinessDay` in the `Impl` interface instead forces all derived classes to negate that logic instead of implementing it directly. It's like having them implement `isNotHoliday`.

Derived calendar classes can provide specialized behavior by defining an inner class derived from `Calendar::Impl`; their constructor will create a shared pointer to an `Impl` instance and store it in the `impl_data` member of the base class. The resulting calendar can be safely copied by any class that need to store a `Calendar` instance; even when sliced, it will maintain the correct behavior thanks to the contained pointer to the polymorphic `Impl` class. Finally, we can note that instances of the same derived calendar class can share the same `Impl` instance. This can be seen as an implementation of the Flyweight pattern—bringing the grand total to about two and a half patterns for one deceptively simple class.

Enough with the implementation of `Calendar`, and back to its behavior. Here's the surprise I mentioned in the previous section. Remember `Period(1, Weeks)` being equal to `Period(7, Days)`? Except that for the `advance` method of a calendar, 7 days means 7 *business* days. Thus, we have a situation in which two periods `p1` and `p2` are equal (that is, `p1 == p2` returns `true`) but `calendar.advance(p1)` differs from `calendar.advance(p2)`. Yay, us.

I'm not sure I have a good idea for a solution here. Since we want backwards compatibility, the current uses of `Days` must keep working in the same way; so it's not possible, say, to start interpreting `calendar.advance(7, Days)` as 7 calendar days. One way out might be to keep the current situation, introduce two new enumeration cases `BusinessDays` and `CalendarDays` that remove the ambiguity, and deprecate `Days`. Another is to just remove the inconsistency by dictating that a 7-days period do not, in fact, equal one week; I'm not overly happy about this one.

As I said, no obvious solution. If you have any other suggestions, I'm all ears.

## Day-count conventions

The `DayCounter` class provides the means to calculate the distance between two dates, either as a number of days or a fraction of an year, according to different conventions. Derived classes such as `Actual360` or `Thirty360` exist; they implement polymorphic behavior by means of the same technique used by the `Calendar` class and described in the previous section.

Unfortunately, the interface has a bit of a rough edge. Instead of just taking two dates, the `yearFraction` method is declared as

```
Time yearFraction(const Date&,
                  const Date&,
                  const Date& refPeriodStart = Date(),
                  const Date& refPeriodEnd = Date()) const;
```

The two optional dates are required by one specific day-count convention (namely, the ISMA actual/actual convention) that requires a reference period to be specified besides the two input dates.

To keep a common interface, we had to add the two additional dates to the signature of the method for all day counters (most of which happily ignore them). This is not the only mischief caused by this day counter; you'll see another in the next section.

## Schedules

The Schedule class, shown in the next listing, is used to generate sequences of coupon dates.

Interface of the **Schedule** class.

---

```

class Schedule {
    public:
        Schedule(const Date& effectiveDate,
                  const Date& terminationDate,
                  const Period& tenor,
                  const Calendar& calendar,
                  BusinessDayConvention convention,
                  BusinessDayConvention terminationDateConvention,
                  DateGeneration::Rule rule,
                  bool endOfMonth,
                  const Date& firstDate = Date(),
                  const Date& nextToLastDate = Date());
        Schedule(const std::vector<Date>&,
                  const Calendar& calendar = NullCalendar(),
                  BusinessDayConvention convention = Unadjusted,
                  ... /* `other optional parameters` */);

        Size size() const;
        bool empty() const;
        const Date& operator[](Size i) const;
        const Date& at(Size i) const;
        const_iterator begin() const;
        const_iterator end() const;

        const Calendar& calendar() const;
        const Period& tenor() const;
        bool isRegular(Size i) const;
        Date previousDate(const Date& refDate) const;
        Date nextDate(const Date& refDate) const;
        ... // other inspectors and utilities
};
```

---

Following practice and ISDA conventions, this class has to accept a lot of parameters; you can see them as the argument list of its constructor. (Oh, and you'll forgive me if I don't go and explain all

of them. I’m sure you can guess what they mean.) They’re probably too many, which is why the library uses the Named Parameter Idiom (already described in [chapter 4](#)) to provide a less unwieldy factory class. With its help, a schedule can be instantiated as

```
Schedule s = MakeSchedule().from(startDate).to(endDate)
    .withFrequency(Semiannual)
    .withCalendar(TARGET())
    .withNextToLastDate(stubDate)
    .backwards();
```

Other methods include on the one hand, inspectors for the stored data; and on the other hand, methods to give the class a sequence interface, e.g., `size`, `operator[]`, `begin`, and `end`.

The `Schedule` class has a second constructor, taking a precomputed vector of dates and a number of optional parameters, which might be passed to help the library use the resulting schedule correctly. Such information includes the date generation rule or whether the dates are aligned to the end of the month, but mainly, you’ll probably need to pass the `tenor` and an `isRegular` vector of bools, about which I need to spend a couple of words.

What does “regular” mean? The boolean `isRegular(i)` doesn’t refer to the `i`-th date, but to the `i`-th interval; that is, the one between the `i`-th and `(i+1)`-th dates. When a schedule is built based on a tenor, most intervals correspond to the passed tenor (and thus are regular) but the first and last intervals might be shorter or longer depending on whether we passed an explicit first or next-to-last date. We might do this, e.g., when we want to specify a short first coupon.

If we build the schedule with a precomputed set of dates, we don’t have the tenor information and we can’t tell if a given interval is regular unless those pieces of information are passed to the schedule.<sup>7</sup> In turn, this means that using that schedule to build a sequence of coupons (by passing it, say, to the constructor of a fixed-rate bond) might give us the wrong result. And why, oh, why does the bond needs this missing info in order to build the coupons? Again, because the day-count convention of the bond might be ISMA actual/actual, which needs a reference period; and in order to calculate the reference period, we need to know the coupon tenor. In absence of this information, all the bond can do is assume that the coupon is regular, that is, that the distance between the passed start and end dates of the coupon also corresponds to its tenor.

## Finance-related classes

Given our domain, it is only to be expected that a number of classes directly model financial concepts. A few such classes are described in this section.

---

<sup>7</sup>Well, we could use heuristics, but it could get ugly fast.

## Market quotes

There are at least two possibilities to model quoted values. One is to model quotes as a sequence of static values, each with an associated timestamp, with the current value being the latest; the other is to model the current value as a quoted value that changes dynamically.

Both views are useful; and in fact, both were implemented in the library. The first model corresponds to the `TimeSeries` class, which I won't describe in detail here; it is basically a map between dates and values, with methods to retrieve values at given dates and to iterate on the existing values, and it was never really used in other parts of the library. The second resulted in the `Quote` class, shown in the following listing.

Interface of the `Quote` class.

---

```
class Quote : public virtual Observable {
public:
    virtual ~Quote() {}
    virtual Real value() const = 0;
    virtual bool isValid() const = 0;
};
```

---

Its interface is slim enough. The class inherits from the `Observable` class, so that it can notify its dependent objects when its value change. It declares the `isValid` method, that tells whether or not the quote contains a valid value (as opposed to, say, no value, or maybe an expired value) and the `value` method, which returns the current value.

These two methods are enough to provide the needed behavior. Any other object whose behavior or value depends on market values (for example, the bootstrap helpers of [chapter 2](#)) can store handles to the corresponding quotes and register with them as an observer. From that point onwards, it will be able to access the current values at any time.

The library defines a number of quotes—that is, of implementations of the `Quote` interface. Some of them return values which are derived from others; for instance, `ImpliedStdDevQuote` turns option prices into implied-volatility values. Others adapt other objects; `ForwardValueQuote` returns forward index fixings as the underlying term structures change, while `LastFixingQuote` returns the latest value in a time series.

At this time, only one implementation is a genuine source of external values; that would be the `SimpleQuote` class, shown in the next listing.

Implementation of the `SimpleQuote` class.

---

```

class SimpleQuote : public Quote {
public:
    SimpleQuote(Real value = Null<Real>())
        : value_(value) {}

    Real value() const {
        QL_REQUIRE(isValid(), "invalid SimpleQuote");
        return value_;
    }

    bool isValid() const {
        return value_!=Null<Real>();
    }

    Real setValue(Real value) {
        Real diff = value-value_;
        if (diff != 0.0) {
            value_ = value;
            notifyObservers();
        }
        return diff;
    }

private:
    Real value_;
};
```

---

It is simple in the sense that it doesn't implement any particular data-feed interface: new values are set manually by calling the appropriate method. The latest value (possibly equal to `Null<Real>()` to indicate no value) is stored in a data member. The `Quote` interface is implemented by having the `value` method return the stored value, and the `isValid` method checking whether it's null. The method used to feed new values is `setValue`; it takes the new value, notifies its observers if it differs from the latest stored one, and returns the increment between the old and new values.<sup>8</sup>

I'll conclude this post with a few short notes. The first is that the type of the quoted values is constrained to `Real`. This has not been a limitation so far, and besides, it's now too late to define `Quote` as a class template; so it's unlikely that this will ever change.

The second is that the original idea was that the `Quote` interface would act as an adapter to actual data feeds, with different implementations calling the different API and allowing QuantLib to use

---

<sup>8</sup>The choice to return the latest increment is kind of unusual; the idiomatic choice in C and C++ would be to return the old value.

them in a uniform way. So far, however, nobody provided such implementations; the closer we got was to use data feeds in Excel and set their values to instances of `SimpleQuote`.

The last (and a bit longer) note is that the interface of `SimpleQuote` might be modified in future to allow more advanced uses. When setting new values to a group of related quotes (say, the quotes interest rates used for bootstrapping a curve) it would be better to only trigger a single notification after all values are set, instead of having each quote send a notification when it's updated. This behavior would be both faster, since chains of notifications turn out to be quite the time sink, and safer, since no observer would risk to recalculate after only a subset of the quotes are updated. The change (namely, an additional `silent` parameter to `setValue` that would mute notifications when equal to `true`) has already been implemented in a fork of the library, and could be added to QuantLib too.

## Interest rates

The `InterestRate` class (shown in the listing that follows) encapsulates general interest-rate calculations. Instances of this class are built from a rate, a day-count convention, a compounding convention, and a compounding frequency (note, though, that the value of the rate is always annualized, whatever the frequency). This allows one to specify rates such as “5%, actual/365, continuously compounded” or “2.5%, actual/360, semiannually compounded.” As can be seen, the frequency is not always needed. I'll return to this later.

Outline of the `InterestRate` class.

---

```

enum Compounding { Simple,           // 1+rT
                    Compounded,        // (1+r)^T
                    Continuous,         // e^{rT}
                    SimpleThenCompounded
};

class InterestRate {
    public:
        InterestRate(Rate r,
                      const DayCounter&,
                      Compounding,
                      Frequency);
        // inspectors
        Rate rate() const;
        const DayCounter& dayCounter();
        Compounding compounding() const;
        Frequency frequency() const;
        // automatic conversion
        operator Rate() const;
        // implied discount factor and compounding after a given time
}
```

```

// (or between two given dates)
DiscountFactor discountFactor(Time t) const;
DiscountFactor discountFactor(const Date& d1,
                             const Date& d2) const;
Real compoundFactor(Time t) const;
Real compoundFactor(const Date& d1,
                     const Date& d2) const;
// other calculations
static InterestRate impliedRate(Real compound,
                                  const DayCounter&,
                                  Compounding,
                                  Frequency,
                                  Time t);
... // same with dates
InterestRate equivalentRate(Compounding,
                            Frequency,
                            Time t) const;
... // same with dates
};

```

---

Besides the obvious inspectors, the class provides a number of methods. One is the conversion operator to `Rate`, i.e., to `double`. On afterthought, this is kind of risky, as the converted value loses any day-count and compounding information; this might allow, say, a simply-compounded rate to slip undetected where a continuously-compounded one was expected. The conversion was added for backward compatibility when the `InterestRate` class was first introduced; it might be removed in a future revision of the library, dependent on the level of safety we want to force on users.<sup>9</sup>

Other methods complete a basic set of calculations. The `compoundFactor` returns the unit amount compounded for a time  $t$  (or equivalently, between two dates  $d_1$  and  $d_2$ ) according to the given interest rate; the `discountFactor` method returns the discount factor between two dates or for a time, i.e., the reciprocal of the compound factor; the `impliedRate` method returns a rate that, given a set of conventions, yields a given compound factor over a given time; and the `equivalentRate` method converts a rate to an equivalent one with different conventions (that is, one that results in the same compounded amount).

Like the `InterestRate` constructor, some of these methods take a compounding frequency. As I mentioned, this doesn't always make sense; and in fact, the `Frequency` enumeration has a `NoFrequency` item just to cover this case.

Obviously, this is a bit of a smell. Ideally, the frequency should be associated only with those compounding conventions that need it, and left out entirely for those (such as `Simple` and `Continuous`) that don't. If C++ supported it, we would write something like

---

<sup>9</sup>There are different views on safety among the core developers, ranging from “babysit the user and don't let him hurt himself” to “give him his part of the inheritance, pat him on his back, and send him to find his place in the world.”

```
enum Compounding { Simple,
                    Compounded(Frequency),
                    Continuous,
                    SimpleThenCompounded(Frequency)
};
```

which would be similar to algebraic data types in functional languages, or case classes in Scala;<sup>10</sup> but unfortunately that's not an option. To have something of this kind, we'd have to go for a full-featured Strategy pattern and turn `Compounding` into a class hierarchy. That would probably be overkill for the needs of this class, so we're keeping both the enumeration and the smell.

## Indexes

Like other classes such as `Instrument` and `TermStructure`, the `Index` class is a pretty wide umbrella: it covers concepts such as interest-rate indexes, inflation indexes, stock indexes—you get the drift.

Needless to say, the modeled entities are diverse enough that the `Index` class has very little interface to call its own. As shown in the following listing, all its methods have to do with index fixings.

Interface of the `Index` class.

---

```
class Index : public Observable {
public:
    virtual ~Index() {}
    virtual std::string name() const = 0;
    virtual Calendar fixingCalendar() const = 0;
    virtual bool isValidFixingDate(const Date& fixingDate)
               const = 0;
    virtual Real fixing(const Date& fixingDate,
                        bool forecastTodaysFixing = false)
               const = 0;
    virtual void addFixing(const Date& fixingDate,
                          Real fixing,
                          bool forceOverwrite = false);
    void clearFixings();
};
```

---

The `isValidFixingDate` method tells us whether a fixing was (or will be made) on a given date; the `fixingCalendar` method returns the calendar used to determine the valid dates; and the `fixing` method retrieves a fixing for a past date or forecasts one for a future date. The remaining methods deal specifically with past fixings: the `name` method, which returns an identifier that must be unique

<sup>10</sup>Both support pattern matching on an object, which is like a neater `switch` on steroids. Go have a look when you have some time.

for each index, is used to index (pun not intended) into a map of stored fixings; the `addFixing` method stores a fixing (or many, in other overloads not shown here); and the `clearFixing` method clears all stored fixings for the given index.

Why the map, and where is it in the `Index` class? Well, we started from the requirement that past fixings should be shared rather than per-instance; if one stored, say, the 6-months Euribor fixing for a date, we wanted the fixing to be visible to all instances of the same index,<sup>11</sup> and not just the particular one whose `addFixing` method we called. This was done by defining and using an `IndexManager` singleton behind the curtains. Smelly? Sure, as all singletons. An alternative might have been to define static class variables in each derived class to store the fixings; but that would have forced us to duplicate them in each derived class with no real advantage (it would be as much against concurrency as the singleton).

Since the returned index fixings might change (either because their forecast values depend on other varying objects, or because a newly available fixing is added and replaces a forecast) the `Index` class inherits from `Observable` so that instruments can register with its instances and be notified of such changes.

At this time, `Index` doesn't inherit from `Observer`, although its derived classes do (not surprisingly, since forecast fixings will almost always depend on some observable market quote). This was not an explicit design choice, but rather an artifact of the evolution of the code and might change in future releases. However, even if we were to inherit `Index` from `Observer`, we would still be forced to have some code duplication in derived classes, for a reason which is probably worth describing in more detail.

I already mentioned that fixings can change for two reasons. One is that the index depends on other observables to forecast its fixings; in this case, it simply registers with them (this is done in each derived class, as each class has different observables). The other reason is that a new fixing might be made available, and that's more tricky to handle. The fixing is stored by a call to `addFixing` on a particular index instance, so it seems like no external notification would be necessary, and that the index can just call the `notifyObservers` method to notify its observers; but that's not the case. As I said, the fixings is shared; if we store today's 3-months Euribor fixing, it will be available to all instances of such index, and thus we want all of them to be aware of the change. Moreover, instruments and curves might have registered with any of those `Index` instances, so all of them must send in turn a notification.

The solution is to have all instances of the same index communicate by means of a shared object; namely, we used the same `IndexManager` singleton that stores all index fixings. As I said, `IndexManager` maps unique index tags to sets of fixings; also, by making the sets instances of the `ObservableValue` class, it provides the means to register and receive notification when one or more fixings are added for a specific tag (this class is described later in this appendix. You don't need the details here).

All pieces are now in place. Upon construction, any `Index` instance will ask `IndexManager` for the

---

<sup>11</sup>Note that by "instances of the same index" I mean here instances of the same specific index, not of the same class (which might group different indexes); for instance, `USDLibor(3*Months)` and `USDLibor(6*Months)` are *not* instances of the same index; two different `USDLibor(3*Months)` are.

shared observable corresponding to the tag returned by its `name` method. When we call `addFixings` on, say, some particular 6-months Euribor index, the fixing will be stored into `IndexManager`; the observable will send a notification to all 6-months Euribor indexes alive at that time; and all will be well with the world.

However, C++ still throws a small wrench in our gears. Given the above, it would be tempting to call

```
registerWith(IndexManager::instance().notifier(name()));
```

in the `Index` constructor and be done with it. However, it wouldn't work; for the reason that in the constructor of the base class, the call to the virtual method `name` wouldn't be polymorphic.<sup>12</sup> From here stems the code duplication I mentioned a few paragraphs earlier; in order to work, the above method call must be added to the constructor of each derived index class which implements or overrides the `name` method. The `Index` class itself doesn't have a constructor (apart from the default one that the compiler provides).

As an example of a concrete class derived from `Index`, the next listing sketches the `InterestRateIndex` class.

Sketch of the `InterestRateIndex` class.

---

```
class InterestRateIndex : public Index, public Observer {
public:
    InterestRateIndex(const std::string& familyName,
                      const Period& tenor,
                      Natural settlementDays,
                      const Currency& currency,
                      const Calendar& fixingCalendar,
                      const DayCounter& dayCounter);
    : familyName_(familyName), tenor_(tenor), ... {
        registerWith(Settings::instance().evaluationDate());
        registerWith(
            IndexManager::instance().notifier(name()));
    }

    std::string name() const;
    Calendar fixingCalendar() const;
    bool isValidFixingDate(const Date& fixingDate) const {
        return fixingCalendar().isBusinessDay(fixingDate);
    }
    Rate fixing(const Date& fixingDate,
```

---

<sup>12</sup>If you're not familiar with the darker corners of C++: when the constructor of a base class is executed, any data members defined in derived classes are not yet built. Since any behavior specific to the derived class is likely to depend on such yet-not-existing data, C++ bails out and uses the base-class implementation of any virtual method called in the base-class constructor body.

```

        bool forecastTodaysFixing = false) const;
void update() { notifyObservers(); }

std::string familyName() const;
Period tenor() const;
... // other inspectors

Date fixingDate(const Date& valueDate) const;
virtual Date valueDate(const Date& fixingDate) const;
virtual Date maturityDate(const Date& valueDate) const = 0;
protected:
virtual Rate forecastFixing(const Date& fixingDate)
                           const = 0;

std::string familyName_;
Period tenor_;
Natural fixingDays_;
Calendar fixingCalendar_;
Currency currency_;
DayCounter dayCounter_;

};

std::string InterestRateIndex::name() const {
    std::ostringstream out;
    out << familyName_;
    if (tenor_ == 1*Days) {
        if (fixingDays_==0) out << "ON";
        else if (fixingDays_==1) out << "TN";
        else if (fixingDays_==2) out << "SN";
        else out << io::short_period(tenor_);
    } else {
        out << io::short_period(tenor_);
    }
    out << " " << dayCounter_.name();
    return out.str();
}

Rate InterestRateIndex::fixing(
                           const Date& d,
                           bool forecastTodaysFixing) const {
QL_REQUIRE(isValidFixingDate(d), ...);
Date today = Settings::instance().evaluationDate();
if (d < today) {
    Rate pastFixing =

```

```

        IndexManager::instance().getHistory(name())[d];
        QL_REQUIRE(pastFixing != Null<Real>(), ...);
        return pastFixing;
    }
    if (d == today && !forecastTodaysFixing) {
        Rate pastFixing = ...;
        if (pastFixing != Null<Real>())
            return pastFixing;
    }
    return forecastFixing(d);
}

Date InterestRateIndex::valueDate(const Date& d) const {
    QL_REQUIRE(isValidFixingDate(d) ...);
    return fixingCalendar().advance(d, fixingDays_, Days);
}

```

---

As you might expect, such class defines a good deal of specific behavior besides what it inherits from `Index`. To begin with, it inherits from `Observer`, too, since `Index` doesn't. The `InterestRateIndex` constructor takes the data needed to specify the index: a family name, as in "Euribor", common to different indexes of the same family such as, say, 3-months and 6-months Euribor; a tenor that specifies a particular index in the family; and additional information such as the number of settlement days, the index currency, the fixing calendar, and the day-count convention used for accrual.

The passed data are, of course, copied into the corresponding data members; then the index registers with a couple of observables. The first is the global evaluation date; this is needed because, as I'll explain shortly, there's a bit of date-specific behavior in the class that is triggered when an instance is asked for today's fixing. The second observable is the one which is contained inside `IndexManager` and provides notifications when new fixings are stored. We can identify this observable here: the `InterestRateIndex` class has all the information needed to determine the index, so it can implement the `name` method and call it. However, this also means that classes deriving from `InterestRateIndex` must not override `name`; since the overridden method would not be called in the body of this constructor (as explained [earlier](#)), they would register with the wrong notifier. Unfortunately, this can't be enforced in C++, which doesn't have a keyword like `final` in Java or `sealed` in C#; but the alternative would be to require that all classes derived from `InterestRateIndex` register with `IndexManager`, which is equally not enforceable, probably more error-prone, and certainly less convenient.

The other methods defined in `InterestRateIndex` have different purposes. A few implement the required `Index` and `Observer` interfaces; the simplest are `update`, which simply forwards any notification, `fixingCalendar`, which returns a copy of the stored calendar instance, and `isValidFixingDate`, which checks the date against the fixing calendar.

The `name` method is a bit more complicated. It stitches together the family name, a short representation of the tenor, and the day-count convention to get an index name such as “Euribor 6M Act/360” or “USD Libor 3M Act/360”; special tenors such as overnight, tomorrow-next and spot-next are detected so that the corresponding acronyms are used.

The `fixing` method contains the most logic. First, the required fixing date is checked and an exception is raised if no fixing was supposed to take place on it. Then, the fixing date is checked against today’s date. If the fixing was in the past, it must be among those stored in the `IndexManager` singleton; if not, an exception is raised since there’s no way we can forecast a past fixing. If today’s fixing was requested, the index first tries looking for the fixing in the `IndexManager` and returns it if found; otherwise, the fixing is not yet available. In this case, as well as for a fixing date in the future, the index forecasts the value of the fixing; this is done by calling the `forecastFixing` method, which is declared as purely virtual in this class and implemented in derived ones. The logic in the `fixing` method is also the reason why, as I mentioned, the index registers with the evaluation date; the behavior of the index depends on the value of today’s date, so it need to be notified when it changes.

Finally, the `InterestRateIndex` class defines other methods that are not inherited from `Index`. Most of them are inspectors that return stored data such as the family name or the tenor; a few others deal with date calculations. The `valueDate` method takes a fixing date and returns the starting date for the instrument that underlies the rate (for instance, the deposit underlying a LIBOR, which for most currencies starts two business days after the fixing date); the `maturityDate` method takes a value date and returns the maturity of the underlying instrument (e.g., the maturity of the deposit); and the `fixingDate` method is the inverse of `valueDate`, taking a value date and returning the corresponding fixing date. Some of these methods are virtual, so that their behavior can be overridden; for instance, while the default behavior for `valueDate` is to advance the given number of fixing days on the given calendar, LIBOR index mandates first to advance on the London calendar, then to adjust the resulting date on the calendar corresponding to the index currency. For some reason, `fixingDate` is not virtual; this is probably an oversight that should be fixed in a future release.

## Aside: how much generalization?

Some of the methods of the `InterestRateIndex` class were evidently designed with LIBOR in mind, since that was the first index of that kind implemented in the library. On the one hand, this makes the class less generic than one would like: for instance, if we were to decide that the 5–10 years swap-rate spread were to be considered an interest-rate index in its own right, we would be hard-pressed to fit it to the interface of the base class and its single `tenor` method. But on the other hand, it is seldom wise to generalize an interface without having a couple of examples of classes that should implement it; and a spread between two indexes (being just that; a spread, not an index) is probably not one such class.

## Exercises and payoffs

I'll close this section with a couple of domain-related classes used in the definitions of a few instruments.

First, the `Exercise` class, shown in the listing below.

Interface of the `Exercise` class and its derived classes.

---

```
class Exercise {
public:
    enum Type {
        American, Bermudan, European
    };
    explicit Exercise(Type type);
    virtual ~Exercise();
    Type type() const;
    Date date(Size index) const;
    const std::vector<Date>& dates() const;
    Date lastDate() const;
protected:
    std::vector<Date> dates_;
    Type type_;
};

class EarlyExercise : public Exercise {
public:
    EarlyExercise(Type type,
                  bool payoffAtExpiry = false);
    bool payoffAtExpiry() const;
};

class AmericanExercise : public EarlyExercise {
public:
    AmericanExercise(const Date& earliestDate,
                     const Date& latestDate,
                     bool payoffAtExpiry = false);
};

class BermudanExercise : public EarlyExercise {
public:
    BermudanExercise(const std::vector<Date>& dates,
                     bool payoffAtExpiry = false);
};
```

---

```
class EuropeanExercise : public Exercise {
    public:
        EuropeanExercise(const Date& date);
};
```

---

As you would expect, the base class declares methods to retrieve information on the date, or dates, of the exercise. Quite a few of them, actually. There's a `dates` method that returns the set of exercise dates, a `date` method that returns the one at a particular index, and a convenience method `lastDate` that, as you might have guessed, returns the last one; so there's some redundancy there.<sup>13</sup> Also, there's a `type` method that is leaving me scratching my head as I look at the code again.

The `type` method returns the kind of exercise, picking its value from a set (European, Bermudan, or American) declared in an inner enumeration `Exercise::Type`. This is not puzzling *per se*, but it goes somewhat against what we did next, which is to use inheritance to declare `AmericanExercise`, `BermudanExercise`, and `EuropeanExercise` classes. On the one hand, the use of a virtual destructor in the base `Exercise` class seems to suggest that inheritance is the way to go if one wants to define new kind of exercises. On the other hand, enumerating the kind of exercises in the base class seems to go against this kind of extension, since inheriting a new exercise class would also require one to add a new case to the enumeration. For inheritance, one can also argue that the established idiom around the library is to pass around smart pointers to the `Exercise` class; and against inheritance, that the class doesn't define any virtual method except the destructor, and the behavior of an instance of any derived class is only given by the value of the data members stored in the base class. In short: it seems that, when we wrote this, we were even more confused than I am now.

Were I to write it now, I'd probably keep the enumeration and make it a concrete class: the derived classes might either create objects that can be safely sliced to become `Exercise` instances when passed around, or they could be turned into functions returning `Exercise` instances directly. As much as this might irk object-oriented purists, there are a number of places in the code where the type of the exercise need to be checked, and having an enumeration is probably the pragmatic choice when compared to using casts or some kind of visitor pattern. The absence of specific behavior in derived classes seems another hint to me.

As I wrote this, it occurred to me that an exercise might also be an `Event`, as described in [chapter 4](#). However, this doesn't always match what the `Exercise` class models. In the case of a European exercise, we could also model it as an `Event` instance; in the case of a Bermudan exercise, the `Exercise` instance would probably correspond to a set of `Event` instances; and in the case of an American exercise, what we're really modeling here is an exercise range—and as a matter of fact, the meaning of the interface also changes in this case, since the `dates` method no longer returns the set of all possible exercise dates, but just the first and last date in the range. As often happens, the small things that seem obvious turn out to be difficult to model soundly when looked up close.

\* \* \*

---

<sup>13</sup>Lovers of encapsulation will probably prefer the version taking an index to the one returning a vector, since the latter reveals more than necessary about the internal storage of the class.

Onwards to the **Payoff** class, shown in the next listing together with a few of its derived classes.

Interface of the **Payoff** class and a few derived classes.

---

```
class Payoff : std::unary_function<Real,Real> {
public:
    virtual ~Payoff() {}
    virtual std::string name() const = 0;
    virtual std::string description() const = 0;
    virtual Real operator()(Real price) const = 0;
    virtual void accept(AcyclicVisitor&);
};

class TypePayoff : public Payoff {
public:
    Option::Type optionType() const;
protected:
    TypePayoff(Option::Type type);
};

class FloatingTypePayoff : public TypePayoff {
public:
    FloatingTypePayoff(Option::Type type);
    Real operator()(Real price) const;
    // more Payoff interface
};

class StrikedTypePayoff : public TypePayoff {
public:
    Real strike() const;
    // more Payoff interface
protected:
    StrikedTypePayoff(Option::Type type,
                      Real strike);
};

class PlainVanillaPayoff : public StrikedTypePayoff {
public:
    PlainVanillaPayoff(Option::Type type,
                       Real strike);
    Real operator()(Real price) const;
    // more Payoff interface
};
```

---

Its interface includes an `operator()`, returning the value of the payoff given the value of the underlying, an `accept` method to support the Visitor pattern, and a couple of inspectors (`name` and `description`) which can be used for reporting—and are probably one too many.

In hindsight, we tried to model the class before having a grasp of enough use cases. Unfortunately, the resulting interface stuck. The biggest problem is the dependency of `operator()` on a single underlying, which excludes payoffs based on multiple underlying values.<sup>14</sup> Another one is the over-reliance on inheritance. For instance, we have a `TypePayoff` class that adds a type (Call or Put, which again might be restrictive) and the corresponding inspector; a `StrikedTypePayoff` which adds a strike; and, finally, `PlainVanillaPayoff`, which models a simple call or put payoff and ends up removed from the `Payoff` class by three levels of inheritance: probably too many, considering that this is used to implement a textbook option and will be looked up by people just starting with the library.

Another misstep we might have made is to add a pointer to a `Payoff` instance to the `Option` class as a data member, with the intent that it should contain the information about the payoff. This led us to classes such as `FloatingTypePayoff`, also shown in the listing. It's used in the implementation of floating lookback options, and stores the information about the type (as in, call or put); but since the strike is fixed at the maturity of the option, it can't specify it and can't implement the payoff with the interface we specified. Its `operator()` throws an exception if invoked. In this case, we might as well do without the payoff and just pass the type to the lookback option; that is, if its base `Option` class didn't expect a payoff.

## Math-related classes

The library also needs some mathematical tools, besides those provided by the C++ standard library. Here is a brief overview of some of them.

### Interpolations

Interpolations belong to a kind of class which is not common in QuantLib: namely, the kind that might be unsafe to use.

The base class, `Interpolation`, is shown in the listing below. It interpolates two underlying random-access sequences of `x` and `y` values, and provides an `operator()` that returns the interpolated values as well as a few other convenience methods. Like the `Calendar` class we saw in [a previous section](#), it implements polymorphic behavior by means of the pimpl idiom: it declares an inner `Impl` class whose derived classes will implement specific interpolations and to which the `Interpolation` class forwards its own method calls. Another inner class template, `templateImpl`, implements the common machinery and stores the underlying data.

---

<sup>14</sup>This also constrains how we model payoffs based on multiple fixings of an underlying; for instance, Asian options are passed the payoff for a plain option and the average of the fixings is done externally, before passing it to the payoff. One might want the whole process to be described as “the payoff”.

Sketch of the `Interpolation` class.

---

```

class Interpolation : public Extrapolator {
    protected:
        class Impl {
            public:
                virtual ~Impl() {}
                virtual void update() = 0;
                virtual Real xMin() const = 0;
                virtual Real xMax() const = 0;
                virtual Real value(Real) const = 0;
                virtual Real primitive(Real) const = 0;
                virtual Real derivative(Real) const = 0;
        };
        template <class I1, class I2>
        class templateImpl : public Impl {
            public:
                templateImpl(const I1& xBegin, const I1& xEnd,
                            const I2& yBegin);
                Real xMin() const;
                Real xMax() const;
            protected:
                Size locate(Real x) const;
                I1 xBegin_, xEnd_;
                I2 yBegin_;
        };
        shared_ptr<Impl> impl_;
    public:
        typedef Real argument_type;
        typedef Real result_type;
        bool empty() const { return !impl_; }
        Real operator()(Real x, bool extrapolate = false) const {
            checkRange(x,extrapolate);
            return impl_->value(x);
        }
        Real primitive(Real x, bool extrapolate = false) const;
        Real derivative(Real x, bool extrapolate = false) const;
        Real xMin() const;
        Real xMax() const;
        void update();
    protected:
        void checkRange(Real x, bool extrapolate) const;
};
```

---

As you can see, `templateImpl` doesn't copy the `x` and `y` values; instead, it just provides a kind of view over them by storing iterators into the two sequences. This is what makes interpolations unsafe: on the one hand, we have to make sure that the lifetime of an `Interpolation` instance doesn't exceed that of the underlying sequences, to avoid pointing into a destroyed object; and on the other hand, any class that stores an `interpolation` instance will have to take special care of copying.

The first requirement is not a big problem. An interpolation is seldom used on its own; it is usually stored as a data member of some other class, together with its underlying data. This takes care of the lifetime issues, as the interpolation and the data live and die together.

The second is not a big problem, either: but whereas the first issue is usually taken care of automatically, this one requires some action on the part of the developer. As I said, the usual case is to have an `Interpolation` instance stored in some class together with its data. The compiler-generated copy constructor for the container class would make new copies of the underlying data, which is correct; but it would also make a new copy of the interpolation that would still be pointing at the original data (since it would store copies of the original iterators). This is, of course, not correct.

To avoid this, the developer of the host class needs to write a user-defined copy constructor that not only copies the data, but also regenerates the interpolation so that it points to the new sequences—which might not be so simple. An object holding an `Interpolation` instance can't know its exact type (which is hidden in the `Impl` class) and thus can't just rebuild it to point somewhere else.

One way out of this would have been to give interpolations some kind of virtual `clone` method to return a new one of the same type, or a virtual `rebind` method to change the underlying iterators once copied. However, that wasn't necessary, as most of the times we already have interpolation traits laying around.

What's that, you say? Well, it's those `Linear` or `LogLinear` classes I've been throwing around while I was explaining interpolated term structures in [chapter 3](#). An example is in the following listing, together with its corresponding interpolation class.

Sketch of the `LinearInterpolation` class and of its traits class.

---

```
template <class I1, class I2>
class LinearInterpolationImpl
    : public Interpolation::templateImpl<I1,I2> {
public:
    LinearInterpolationImpl(const I1& xBegin, const I1& xEnd,
                           const I2& yBegin)
        : Interpolation::templateImpl<I1,I2>(xBegin, xEnd, yBegin),
          primitiveConst_(xEnd-xBegin), s_(xEnd-xBegin) {}
    void update();
    Real value(Real x) const {
        Size i = this->locate(x);
        return this->yBegin_[i] + (x-this->xBegin_[i])*s_[i];
    }
}
```

```

    Real primitive(Real x) const;
    Real derivative(Real x) const;
private:
    std::vector<Real> primitiveConst_, s_;
};

class LinearInterpolation : public Interpolation {
public:
    template <class I1, class I2>
    LinearInterpolation(const I1& xBegin, const I1& xEnd,
                        const I2& yBegin) {
        impl_ = shared_ptr<Interpolation::Impl>(
            new LinearInterpolationImpl<I1,I2>(xBegin, xEnd,
                                                yBegin));
        impl_->update();
    }
};

class Linear {
public:
    template <class I1, class I2>
    Interpolation interpolate(const I1& xBegin, const I1& xEnd,
                             const I2& yBegin) const {
        return LinearInterpolation(xBegin, xEnd, yBegin);
    }
    static const bool global = false;
    static const Size requiredPoints = 2;
};

```

---

The `LinearInterpolation` class doesn't have a lot of logic: its only method is a template constructor (the class itself is not a template) that instantiates an inner implementation class. The latter inherits from `TemplateImpl` and is the one that does the heavy lifting, implementing the actual interpolation formulas (with the help of methods, such as `locate`, defined in its base class).

The `Linear` traits class defines some static information, namely, that we need at least two points for a linear interpolation, and that changing a point only affects the interpolation locally; and also defines an `interpolate` method that can create an interpolation of a specific type from a set of iterators into `x` and `y`. The latter method is implemented with the same interface by all traits (when an interpolation, such as splines, needs more parameters, they are passed to the traits constructor and stored) and is the one that's going to help in our copying problem. If you look, for instance, at the listing of the `InterpolatedZeroCurve` class back in [chapter 3](#), you'll see that we're storing an instance of the traits class (it's called `interpolator_` there) together with the interpolation and the underlying data. If we do the same in any class that stores an interpolation, we'll be able to use the

traits to create a new one in the copy constructor.

Unfortunately, though, we have no way at this time to enforce writing a copy constructor in a class that stores an interpolation, so its developer will have to remember it. We have no way, that is, without making the `Interpolation` class non-copyable and thus also preventing useful idioms (like returning an interpolation from a method, as traits do). In C++11, we'd solve this by making it non-copyable and movable.

### Aside: gordian knots

When implementing a class that stores an interpolation, an alternative to writing a copy constructor would be to cut right through the problem and make the class non-copyable. It might be less of a problem than it seems: such classes are usually term structures, and more often than not they're passed around inside `shared_ptrs`, which don't require copying.

(True story: most curves have been non-copyable for a while before release 1.0, and nobody complained much about it. Eventually, we reintroduced copying as a convenience, but I'm still not sure it was necessary.)

A final note: the interpolation stores iterators into the original data, but this is not enough to keep it up to date when any of the data changes. When this happens, its `update` method must be called so that the interpolation can refresh its state; this is the responsibility of the class that contains the interpolation and the data (and which, probably, is registered as an observer with whatever may change.) This holds also for those interpolations, such as linear, that might just read the data directly: depending on the implementation, they might precalculate some results and store them as state to be kept updated. (The current `LinearInterpolation` implementation does this for the slopes between the points, as well as the values of its primitive at the nodes.<sup>15</sup> Depending on how frequently the data are updated, this might be either an optimization or a pessimization.)

## One-dimensional solvers

Solvers were used in the bootstrap routines described in chapter 3, the yield calculations mentioned in chapter 4, and any code that needs a calculated value to match a target; i.e., that needs, given a function  $f$ , to find the  $x$  such that  $f(x) = \xi$  within a given accuracy.

The existing solvers will find the  $x$  such that  $f(x) = 0$ ; of course, this doesn't make them any less generic, but it requires you to define the additional helper function  $g(x) \equiv f(x) - \xi$ . There are a few of them, all based on algorithms which were taken from *Numerical Recipes in C* (Press *et al*, 1992) and duly reimplemented.<sup>16</sup>

<sup>15</sup>The presence of the `primitive` and `derivative` methods is a bit of implementation leak. They were required by interpolated interest-rate curves in order to pass from zero rates to forwards and back.

<sup>16</sup>This also goes for a few multi-dimensional optimizers. In that case, apart from the obvious copyright issues, we also rewrote them in order to use idiomatic C++ and start indexing arrays from 0.

The following listing shows the interface of the class template `Solver1D`, used as a base by the available solvers.

Interface of the `Solver1D` class template and of a few derived classes.

---

```

template <class Impl>
class Solver1D : public CuriouslyRecurringTemplate<Impl> {
public:
    template <class F>
    Real solve(const F& f,
               Real accuracy,
               Real guess,
               Real step) const;
    template <class F>
    Real solve(const F& f,
               Real accuracy,
               Real guess,
               Real xMin,
               Real xMax) const;
    void setMaxEvaluations(Size evaluations);
    void setLowerBound(Real lowerBound);
    void setUpperBound(Real upperBound);
};

class Brent : public Solver1D<Brent> {
public:
    template <class F>
    Real solveImpl(const F& f,
                  Real xAccuracy) const;
};

class Newton : public Solver1D<Newton> {
public:
    template <class F>
    Real solveImpl(const F& f,
                  Real xAccuracy) const;
};

```

---

It provides some boilerplate code, common to all of them: one overload of the `solve` method looks for lower and upper values of  $x$  that bracket the solution, while the other checks that the solution is actually bracketed by the passed minimum and maximum values. In both cases, the actual calculation is delegated to the `solveImpl` method defined by the derived class and implementing a specific algorithm. Other methods allow you to set constraints on the explored range, or the number of function evaluations.

The forwarding to `solveImpl` is implemented using the Curiously Recurring Template Pattern, already described in chapter 7. When we wrote these classes, we were at the height of our template craze (did I mention we even had an implementation of expression templates? (Veldhuizen, 2000)) so you might suspect that the choice was dictated by the fashion of that time. However, it wouldn't have been possible to use dynamic polymorphism. We wanted the solvers to work with any function pointer or function object, and `boost::function` wasn't around yet, which forced us to use a template method. Since the latter couldn't be virtual, CRTP was the only way to put the boilerplate code in the base class and let it call a method defined in derived ones.

A few notes to close this subsection. First: if you want to write a function that takes a solver, the use of CRTP forces you to make it a template, which might be awkward. To be honest, most of the times we didn't bother and just hard-coded an explicit choice of solver. I won't blame you if you do the same. Second: most solvers only use the passed `f` by calling `f(x)`, so they work with anything that can be called as a function, but `Newton` and `NewtonSafe` also require that `f.derivative(x)` be defined. This, too, might have been awkward if we used dynamic polymorphism. Third, and last: the `Solver1D` interface doesn't specify if the passed accuracy  $\epsilon$  should apply to  $x$  (that is, if the returned  $\tilde{x}$  should be within  $\epsilon$  of the true root) or to  $f(x)$  (that is, if  $f(\tilde{x})$  should be within  $\epsilon$  of 0). However, all existing solvers treat it as the accuracy on  $x$ .

## Optimizers

Multi-dimensional optimizers are more complex than 1-D solvers. In a nutshell, they find the set of variables  $\tilde{\mathbf{x}}$  for which the cost function  $f(\mathbf{x})$  returns its minimum value; but of course, there's a bit more to it.

Unlike solvers, optimizers don't use templates. They inherit from the base class `OptimizationMethod`, shown in the next listing.

Interface of the `OptimizationMethod` class.

---

```
class OptimizationMethod {
public:
    virtual ~OptimizationMethod() {}
    virtual EndCriteria::Type minimize(
        Problem& P,
        const EndCriteria& endCriteria) = 0;
};
```

---

Its only method, apart from the virtual destructor, is `minimize`. The method takes a reference to a `Problem` instance, which in turn contains references to the function to minimize and to any constraints, and performs the calculations; at the end of which, it has the problem of having too many things to return. Besides the best-solution array  $\tilde{\mathbf{x}}$ , it must return the reason for exiting the calculation (did it converge, or are we returning our best guess after the maximum number of evaluations?) and it would be nice to return  $f(\tilde{\mathbf{x}})$  as well, since it's likely that it was already calculated.

In the current implementation, the method returns the reason for exiting and stores the other results inside the `Problem` instance. This is also the reason for passing the problem as a non-`const` reference; an alternative solution might have been to leave the `Problem` instance alone and to return all required values in a structure, but I see how this might be seen as more cumbersome. On the other hand, I see no reason for `minimize` itself to be non-`const`: my guess is that it was an oversight on our part (I'll get back to this later).

Onward to the `Problem` class, shown in the listing below.

Interface of the `Problem` class.

---

```
class Problem {
public:
    Problem(CostFunction& costFunction,
             Constraint& constraint,
             const Array& initialValue = Array());

    Real value(const Array& x);
    Disposable<Array> values(const Array& x);
    void gradient(Array& grad_f, const Array& x);
    // ... other calculations ...

    Constraint& constraint() const;
    // ... other inspectors ...

    const Array& currentValue();
    Real functionValue() const;
    void setCurrentValue(const Array& currentValue);
    Integer functionEvaluation() const;
    // ... other results ...
};
```

---

As I mentioned, it groups together arguments such as the cost function to minimize, the constraints, and an optional guess. It provides methods that call the underlying cost function while keeping track of the number of evaluation, and that I'll describe when talking about cost functions; a few inspectors for its components; and methods to retrieve the results (as well as to set them; the latter are used by the optimizers).

The problem (no pun intended) is that it takes and stores its components as non-`const` references. I'll talk later about whether they might be `const` instead. The fact that they're reference is an issue in itself, since it puts the responsibility on client code to make sure that their lifetimes last at least as much as that of the `Problem` instance.

In an alternative implementation in which the optimizer returned its results in a structure, the issue might be moot: we might do away with the `Problem` class and pass its components directly to

the `minimize` method. This would sidestep the lifetime issue, since they wouldn't be stored. The disadvantage would be that each optimizer would have to keep track of the number of function evaluations, causing some duplication in the code base.

Also unlike for 1-D solvers, the cost function is not a template parameter for the minimization method. It needs to inherit from the `CostFunction` class, shown in the next listing.

Interface of the `CostFunction` class.

---

```
class CostFunction {
public:
    virtual ~CostFunction() {}

    virtual Real value(const Array& x) const = 0;
    virtual Array values(const Array& x) const = 0;

    virtual void gradient(Array& grad, const Array& x) const;
    virtual Real valueAndGradient(Array& grad,
                                  const Array& x) const;
    virtual void jacobian(Matrix &jac, const Array &x) const;
    virtual Array valuesAndJacobian(Matrix &jac,
                                    const Array &x) const;
};


```

---

Unsurprisingly, its interface declares the `value` method, which returns, well, the value of the function for the given array of arguments;<sup>17</sup> but it also declares a `values` method returning an array, which is a bit more surprising until you remember that optimizers are often used for calibrating over a number of quotes. Whereas `value` returns the total error to minimize (let's say, the sum of the squares of the errors, or something like it), `values` returns the set of errors over each quote; there are algorithms that can make use of this information to converge more quickly.

Other methods return the derivatives of the value, or the values, for use by some specific algorithms: `gradient` calculates the derivative of `value` with respect to each variable and stores it in the array passed as first argument, `jacobian` does the same for `values` filling a matrix, and the `valueAndGradient` and `valuesAndJacobian` methods calculate both values and derivatives at the same time for efficiency. They have a default implementation that calculates numerical derivatives; but of course that's costly, and derivative-based algorithms should only be used if you can override the method with an analytic calculation.

A note: checking the interface of `CostFunction` shows that all its methods are declared as `const`, so passing it as non-`{const}` reference to the `Problem` constructor was probably a goof. Changing it to `const` would widen the contract of the constructor, so it should be possible without breaking backward compatibility.

Finally, the `Constraint` class, shown in the following listing.

---

<sup>17</sup>Although you might be a bit surprised that it doesn't declare `operator()` instead.

Interface of the `Constraint` class.

---

```
class Constraint {
protected:
    class Impl;
public:
    bool test(const Array& p) const;
    Array upperBound(const Array& params) const;
    Array lowerBound(const Array& params) const;
    Real update(Array& p,
                const Array& direction,
                Real beta);
    Constraint(const shared_ptr<Impl>& impl =
               shared_ptr<Impl>());
};
```

---

It works as a base class for constraints to be applied to the domain of the cost function; the library also defines a few predefined ones, not shown here, as well as a `CompositeConstraint` class that can be used to merge a number of them into a single one.

Its main method is `test`, which takes an array of variables and returns whether or not they satisfy the constraint; that is, if the array belongs to the domain we specified as valid. It also defines the `upperBound` and `lowerBound` methods, which in theory should specify the maximum and minimum value of the variables but in practice can't always specify them correctly; think of the case in which the domain is a circle, and you'll see that there are cases in which  $x$  and  $y$  are both between their upper and lower bounds but the resulting point is outside the domain.

A couple more notes. First, the `Constraint` class also defines an `update` method. It's not `const`, which would make sense if it updated the constraint; except it doesn't. It takes an array of variables and a direction, and extends the original array in the given direction until it satisfies the constraint. It should have been `const`, it should have been named differently, and as the kids say I can't even. This might be fixed, though. Second, the class uses the pimpl idiom (see [a previous section](#)) and the default constructor also takes an optional pointer to the implementation. Were I to write the class today, I'd have a default constructor taking no arguments and an additional constructor taking the implementation and declared as protected and to be used by derived classes only.

Some short final thoughts on the `const`-correctness of these classes. In summary: it's not great, with some methods that can be fixed and some others that can't. For instance, changing the `minimize` method would break backwards compatibility (since it's a virtual method, and constness is part of its signature) as well as a few optimizers, that call other methods from `minimize` and use data members as a way to transfer information between methods.<sup>18</sup> We could have avoided this if we had put some more effort in reviewing code before version 1.0. Let this be a lesson for you, young coders.

---

<sup>18</sup>Some of them declare those data members as `mutable`, suggesting the methods might have been `const` in the past. As I write this, I haven't investigated this further.

## Statistics

The statistics classes were written mostly to collect samples from Monte Carlo simulations (you'll remember them from [chapter 6](#)). The full capabilities of the library are implemented as a series of decorators, each one adding a layer of methods, instead of a monolith; as far as I can guess, that's because you can choose one of two classes at the bottom of the whole thing. A layered design gives you the possibility to build more advanced capabilities just once, based on the common interface of the bottom layer.

The first class you can choose, shown below, is called `IncrementalStatistics`, and has the interface you would more or less expect: it can return the number of samples, their combined weight in case they were weighed, and a number of statistics results.

Interface of the `IncrementalStatistics` class.

---

```
class IncrementalStatistics {
public:
    typedef Real value_type;
    IncrementalStatistics();

    Size samples() const;
    Real weightSum() const;
    Real mean() const;
    Real variance() const;
    Real standardDeviation() const;
    Real errorEstimate() const;
    // skewness, kurtosis, min, max...

    void add(Real value, Real weight = 1.0);
    template <class DataIterator>
    void addSequence(DataIterator begin, DataIterator end);
};
```

---

Samples can be added one by one or as a sequence delimited by two iterators. The shtick of this class is that it doesn't store the data it gets passed, but instead it updates the statistics on the fly; the idea was to save the memory that would be otherwise used for the storage, and in the year 2000 (when a computer might have 128 or 256 MB of RAM) it was a bigger concern than it is now. The implementation used to be homegrown; nowadays it's written in terms of the Boost accumulator library.

The second class, `GeneralStatistics`, implements the same interface and adds a few other methods, made possible by the fact that it stores (and thus can return) the passed data; for instance, it can return percentiles or sort its data. It also provides a template `expectationValue` method that can be used for bespoke calculations; if you're interested, there's more on that in the aside at the end of this section.

Interface of the `GeneralStatistics` class.

---

```
class GeneralStatistics {
public:
    // ... same as IncrementalStatistics ...

    const std::vector<std::pair<Real,Real> >& data() const;

    template <class Func, class Predicate>
    std::pair<Real,Size>
    expectationValue(const Func& f,
                     const Predicate& inRange) const;

    Real percentile(Real y) const;
    // ... other inspectors ...

    void sort() const;
    void reserve(Size n) const;
};
```

---

Next, the outer layers. The first ones add statistics associated with risk, like expected shortfall or value at risk; the problem being that you usually need the whole set of samples for those. In the case of incremental statistics, therefore, we have to forgo the exact calculation and look for approximations. One possibility is to take the mean and variance of the samples, suppose they come from a Gaussian distribution with the same moments, and get analytic results based on this assumption; that's what the `GenericGaussianStatistics` class does.

Interface of the `GaussianStatistics` class.

---

```
template<class S>
class GenericGaussianStatistics : public S {
public:
    typedef typename S::value_type value_type;
    GenericGaussianStatistics() {}
    GenericGaussianStatistics(const S& s) : S(s) {}

    Real gaussianDownsideVariance() const;
    Real gaussianDownsideDeviation() const;
    Real gaussianRegret(Real target) const;
    Real gaussianPercentile(Real percentile) const;
    Real gaussianValueAtRisk(Real percentile) const;
    Real gaussianExpectedShortfall(Real percentile) const;
    // ... other measures ...
};
```

---

---

```
typedef GenericGaussianStatistics<GeneralStatistics>
    GaussianStatistics;
```

---

As I mentioned, and as you can see, it's implemented as a decorator; it takes the class to decorate as a template parameter, inherits from it so that it still has all the methods of its base class, and adds the new methods. The library provides a default instantiation, `GaussianStatistics`, in which the template parameter is `GeneralStatistics`. Yes, I would have expected the incremental version, too, but there's a reason for this; bear with me for a minute.

When the base class stores the full set of samples, we can write a decorator that calculates the actual risk measures; that would be the `GenericRiskStatistics` class. As for the Gaussian statistics, I won't discuss the implementation (you can look them up in the library).

Interface of the `RiskStatistics` class.

---

```
template <class S>
class GenericRiskStatistics : public S {
public:
    typedef typename S::value_type value_type;

    Real downsideVariance() const;
    Real downsideDeviation() const;
    Real regret(Real target) const;
    Real valueAtRisk(Real percentile) const;
    Real expectedShortfall(Real percentile) const;
    // ... other measures ...
};

typedef GenericRiskStatistics<GaussianStatistics>
    RiskStatistics;
```

---

As you can see, the layers can be combined; the default instantiation provided by the library, `RiskStatistics`, takes `GaussianStatistics` as its base and thus provides both Gaussian and actual measures. This was also the reason why `GeneralStatistics` was used as the base for the latter.

On top of it all, it's possible to have other decorators; the library provides a few, but I won't show their code here. One is `SequenceStatistics`, that can be used when a sample is an array instead of a single number, uses internally a vector of instances of scalar statistics classes, and also adds the calculation of the correlation and covariance between the elements of the samples; it is used, for instance, in the LIBOR market model, where each sample usually collects cash flows at different times. Other two are `ConvergenceStatistics` and `DiscrepancyStatistics`; they provide information on the properties of the sequence of samples, aren't used anywhere else in the library, but at least we had the decency of writing unit tests for both of them.

## Aside: extreme expectations.

Looking back at the `GeneralStatistics` class, I'm not sure if I should be proud or ashamed of it, because—oh boy, I really went to town with generalization there.

It might have been the mathematics. It started with a straightforward implementation; but looking at the formulas for the mean, the variance, and even some more complex one defined in other layers, I saw that they all could be written (give or take some later adjustments) as

$$\frac{\sum_{x_i \in \mathcal{R}} f(x_i) w_i}{\sum_{x_i \in \mathcal{R}} w_i},$$

that is, the expected value of  $f(x)$  over some range  $\mathcal{R}$ . For the mean,  $f(x)$  would be the identity and  $\mathcal{R}$  would be the full domain of the samples; for the variance,  $f(x)$  would be  $(x - \bar{x})^2$  over the same range; and so on. The result was a template `expectationValue` method that would take the function  $f$  and the range  $\mathcal{R}$  and return the corresponding result and the number of samples in the range; most other methods are implemented by calling it with the relevant inputs. If you're a bit confused at first about the mean being implemented as

```
return expectationValue(identity<Real>(),
    everywhere()).first;
```

then I can't blame you. By the way, I must have been learning about functional programming at the time; the range is passed as a function that takes a sample and return `true` or `false` depending on whether it's in the range, and `everywhere` above is one of a few small predefined helper functions I added to write this kind of code. It's all fun and games until someone writes  $(x - \bar{x})^2$  as

```
compose(square<Real>(),
    std::bind2nd(std::minus<Real>(), mean()))
```

Self-snark aside: the above is very general and allows client code to create new calculations, but probably caters a bit too much to the math-inclined and can make for cryptic code, so I'm not sure that I stroke the right balance here. Replacing binds with lambdas in C++11 might certainly help; we'll see how this code turns out when we start using them. On the other hand, performance shouldn't be a problem: `expectationValue` is a template, and so are the functions like `compose` and `everywhere` above, so the compiler can see their implementation and can probably inline them. In that case, the result would be the simpler loop we might have written in a direct implementation of the mean or variance formulas.

## Linear algebra

I don't have a lot to write about the current implementation of the `Array` and `Matrix` classes, shown in the following listing.

Sketch of the **Array** and **Matrix** classes.

---

```

class Array {
    public:
        explicit Array(Size size = 0);
        // ... other constructors ...
        Array(const Array&);
        Array(const Disposable<Array>&);
        Array& operator=(const Array&);
        Array& operator=(const Disposable<Array>&);

        const Array& operator+=(const Array&);
        const Array& operator+=(Real);
        // ... other operators ...

        Real operator[](Size) const;
        Real& operator[](Size);

        void swap(Array&);
        // ... iterators and other utilities ...
    private:
        boost::scoped_array<Real> data_;
        Size n_;
};

Disposable<Array> operator+(const Array&, const Array&);
Disposable<Array> operator+(const Array&, Real);
// ... other operators and functions ...

class Matrix {
    public:
        Matrix(Size rows, Size columns);
        // ... other constructors, assignment operators etc. ...

        const_row_iterator operator[](Size) const;
        row_iterator operator[](Size);
        Real& operator()(Size i, Size j) const;

        // ... iterators and other utilities ...
};

Disposable<Matrix> operator+(const Matrix&, const Matrix&);
// ... other operators and functions ...

```

---

Their interface is what you would expect: constructors and assignment operators, element access (the `Array` class provides the `a[i]` syntax; the `Matrix` class provides both `m[i][j]` and `m(i,j)`, because we aim to please), a bunch of arithmetic operators, all working element by element as usual,<sup>19</sup> and a few utilities. There are no methods for resizing, or for other operations suited for containers, because this classes are not meant to be used as such; they're mathematical utilities. Storage is provided by a `scoped_ptr`, which manages the lifetime of the underlying memory.

In the case of `Array`, we also provide a few functions such as `Abs`, `Log` and their like; being good citizens, we're not overloading the corresponding functions in namespace `std` because that's forbidden by the standard. More complex functionality (such as matricial square root, or various decompositions) can be found in separate modules.

In short, a more or less straightforward implementation of arrays and matrices. The one thing which is not obvious is the presence of the `Disposable` class template, which I'll describe in more detail in a further section of this appendix; for the time being, let me just say that it's a pre-C++11 attempt at move semantics.

The idea was to try and reduce the abstraction penalty. Operator overloading is very convenient—after all, `c = a+b` is much easier to read than understand than `add(a,b,c)`—but doesn't come for free: declaring addition as

```
Array operator+(const Array& a, const Array& b);
```

means that the operator must create and return a new `Array` instance, that is, must allocate and possibly copy its memory. When the number of operations increases, so does the overhead.

In the first versions of the library, we tried to mitigate the problem by using expression templates. The idea (that I will only describe very roughly, so I suggest you read ([Veldhuizen, 2000](#)) for details) is that operators don't return an array, but some kind of parse tree holding references to the terms of the expression; so, for instance, `2*a+b` won't actually perform any calculation but only create a small structure with the relevant information. It is only when assigned that the expression is unfolded; and at that point, the compiler would examine the whole thing and generate a single loop that both calculates the result and copies it into the array being assigned.

The technique is still relevant today (possibly even more so, given the progress in compiler technology) but we abandoned it after a few years. Not all compilers were able to process it, forcing us to maintain both expression templates and the simpler implementation, and it was difficult to read and maintain (compare the current declaration of `operator+` with

```
VectorialExpression<
    BinaryVectorialExpression<
        Array::const_iterator, Array::const_iterator, Add> >
operator+(const Array& v1, const Array& v2);
```

---

<sup>19</sup>It always bothered me that `a*b` returns the element-wise product and not the dot product, but I seem to be alone among programmers.

for a taste of what the code was like); therefore, when the C++ community started talking of move semantics and some ideas for implementations began to appear, we took the hint and switched to `Disposable`.

As I said, compilers progressed a lot during these years; nowadays, I'm guessing that all of them would support an expression-template implementation, and the technique itself has probably improved. However, if I were to write the code today (or if I started to change things) the question might be whether to write classes such as `Array` or `Matrix` at all. At the very least, I'd consider implementing them in terms of `std::valarray`, which is supposed to provide facilities for just such a task. In the end, though, I'd probably go for some existing library such as uBLAS: it is available in Boost, it's written by actual experts in numerical code, and we already use it in some parts of the library for specialized calculations.

## Global settings

The `Settings` class, outlined in the listing below, is a singleton (see later) that holds information global to the whole library.

Outline of the `Settings` class.

---

```
class Settings : public Singleton<Settings> {
private:
    class DateProxy : public ObservableValue<Date> {
        DateProxy();
        operator Date() const;
        ...
    };
    ... // more implementation details
public:
    DateProxy& evaluationDate();
    const DateProxy& evaluationDate() const;
    boost::optional<bool>& includeTodaysCashFlows();
    boost::optional<bool> includeTodaysCashFlows() const;
    ...
};
```

---

Most of its data are flags that you can look up in the official documentation, or that you can simply live without; the one piece of information that you'll need to manage is the evaluation date, which defaults to today's date and is used for the pricing of instruments and the fixing of any other quantity.

This poses a challenge: instruments whose value can depend on the evaluation date must be notified when the latter changes. This is done by returning the corresponding information indirectly, namely, wrapped inside a proxy class; this can be seen from the signature of the relevant methods. The proxy

inherits from the `ObservableValue` class template (outlined in the next listing) which is implicitly convertible to `Observable` and overloads the assignment operator in order to notify any changes. Finally, it allows automatic conversion of the proxy class to the wrapped value.

Outline of the `ObservableValue` class template.

---

```
template <class T>
class ObservableValue {
public:
    // initialization and assignment
    ObservableValue(const T& t)
        : value(t), observable_(new Observable) {}
    ObservableValue<T>& operator=(const T& t) {
        value_ = t;
        observable_->notifyObservers();
        return *this;
    }
    // implicit conversions
    operator T() const { return value_; }
    operator shared_ptr<Observable>() const {
        return observable_;
    }
private:
    T value_;
    shared_ptr<Observable> observable_;
};
```

---

This allows one to use the facility with a natural syntax. On the one hand, it is possible for an observer to register with the evaluation date, as in:

```
registerWith(Settings::instance().evaluationDate());
```

on the other hand, it is possible to use the returned value just like a `Date` instance, as in:

```
Date d2 =
    calendar.adjust(Settings::instance().evaluationDate());
```

which triggers an automatic conversion; and on the gripping hand, an assignment can be used for setting the evaluation date, as in:

```
Settings::instance().evaluationDate() = d;
```

which will cause all observers to be notified of the date change.

\* \* \*

Of course, the elephant in the room is the fact that we have a global evaluation date at all. The obvious drawback is that one can't perform two parallel calculations with two different evaluation dates, at least in the default library configuration; but while this is true, it is also not the whole story. On the one hand, there's a compilation flag that allows a program to have one distinct `Settings` instance per thread (with a bit of work on the part of the user) but as we'll see, this doesn't solve all the issues. On the other hand, the global data may cause unpleasantness even in a single-threaded program: even if one wanted to evaluate just an instrument on a different date, the change will trigger recalculation for every other instrument in the system when the evaluation date is set back to its original value.

This clearly points (that is, quite a few smart people had the same idea when we talked about it) to some kind of context class that should replace the global settings. But how would one select a context for any given calculation?

It would be appealing to add a `setContext` method to the `Instrument` class, and to arrange things so that during calculation the instrument propagates the context to its engine and in turn to any term structures that need it. However, I don't think this can be implemented easily.

First, the instrument and its engine are not always aware of all the term structures that are involved in the calculation. For instance, a swap contains a number of coupons, any of which might or might not reference a forecast curve. We're not going to reach them unless we add the relevant machinery to all the classes involved. I'm not sure that we want to set a context to a coupon.

Second, and more important, setting the context for an engine would be a mutating operation. Leaving it to the instrument during calculations would execute it at some point during the call to its `NPV` method, which is supposed to be `const`. This would make it way too easy to trigger a race condition; for instance with a harmless-looking operation such as using the same discount curve for two instruments and evaluating them at different dates. If you have a minimum of experience in parallel programming, you wouldn't dream of, say, relinking the same handle in two concurrent threads; but when the mutation is hidden inside a `const` method, you might not be aware of it. (But wait, you say. Aren't there other mutating operations possibly being done during the call to `NPV`? Good catch: see the aside [at the end of this section](#).)

So it seems that we have to set up the context before starting the calculation. This rules out driving the whole thing from the instrument (because, again, we would be hiding the fact that setting a context to an instrument could undo the work done by another that shared a term structure with the first) and suggests that we'd have to set the context explicitly on the several term structures. On the plus side, we no longer run the risk of a race in which we unknowingly try to set the same context to the same object. The drawbacks are that our setup just got more complex, and that we'd have to duplicate curves if we want to use them concurrently in different contexts: two parallel calculations

on different dates would mean, for instance, two copies of the overnight curve for discounting. And if we have to do this, we might as well manage with per-thread singletons.

Finally, I'm skipping over the scenario in which the context is passed but not saved. It would lead to method calls like

```
termStructure->discount(t, context);
```

which would completely break caching, would cause discomfort to all parties involved, and if we wanted stuff like this we'd write in Haskell.

To summarize: I hate to close the section on a gloomy note, but all is not well. The global settings are a limitation, but I don't have a solution; and what's worse, the possible changes increase complexity. We would not only tell first-time users looking for the Black-Scholes formula that they needs term structures, quotes, an instrument and an engine: we'd also put contexts in the mix. A little help here?

### Aside: more mutations than in a B-movie.

Unfortunately, there are already a number of things that change during a call to the supposedly `const` method `Instrument::NPV`.

To begin with, there are the `arguments` and `results` structures inside the engine, which are read and written during calculation and thus prevent the same engine to be used concurrently for different instruments. This might be fixed by adding a lock to the engine (which would serialize the calculations) or by changing the interface so that the engine's `calculate` method takes the `arguments` structure as a parameter and returns the `results` structure.

Then, there are the `mutable` data members of the instrument itself, which are written at the end of the calculation. Whether this is a problem depends on the kind of calculations one's doing. I suppose that calculating the value of the instrument twice in concurrent threads might just result in the same values being written twice.

The last one that comes to mind is a hidden mutation, and it's probably the most dangerous. Trying to use a term structure during the calculation might trigger its bootstrap, and two concurrent ones would trash each other's calculations. Due to the recursive nature of the bootstrap, I'm not even sure how we could add a lock around it. So if you do decide to perform concurrent calculations (being careful, setting up everything beforehand and using the same evaluation date) be sure to trigger a full bootstrap of your curves before starting.

## Utilities

In QuantLib, there are a number of classes and functions which don't model a financial concept. They are nuts and bolts, used to build some of the scaffolding for the rest of the library. This section is devoted to some of these facilities.

## Smart pointers and handles

The use of run-time polymorphism dictates that many, if not most, objects be allocated on the heap. This raises the problem of memory management—a problem solved in other languages by built-in garbage collection, but left in C++ to the care of the developer.

I will not dwell on the many issues in memory management, especially since they are now mostly a thing of the past. The difficulty of the task (especially in the presence of exceptions) was enough to discourage manual management; therefore, ways were found to automate the process.

The weapons of choice in the C++ community came to be smart pointers: classes that act like built-in pointers but that can take care of the survival of the pointed objects while they are still needed and of their destruction when this is no longer the case. Several implementations of such classes exist which use different techniques; we chose the smart pointers from the Boost libraries (most notably `shared_ptr`, now included in the ANSI/ISO C++ standard). Don't look for `boost::shared_ptr` in the library, though: the relevant classes are imported in an internal QuantLib namespace and used, e.g., as `ext::shared_ptr`. This will allow us to switch to the C++11 implementation painlessly when the time comes.

I won't go into details on shared pointers; you can browse the Boost site for documentation. Here, I'll just mention that their use in QuantLib completely automated memory management. Objects are dynamically allocated all over the place; however, there is not one single `delete` statement in all the tens of thousands of lines of which the library consists.

Pointers to pointers (if you need a quick refresher, see the aside at the end of the section for their purpose and semantics) were also replaced by smart equivalents. We chose not to just use smart pointers to smart pointers; on the one hand, because having to write

```
ext::shared_ptr<ext::shared_ptr<YieldTermStructure> >
```

gets tiresome very quickly—even in Emacs; on the other hand, because the inner `shared_ptr` would have to be allocated dynamically, which just didn't feel right; and on the gripping hand, because it would make it difficult to implement observability. Instead, a class template called `Handle` was provided for this purpose. Its implementation, shown in the listing that follows, relies on an intermediate inner class called `Link` which stores a smart pointer. In turn, the `Handle` class stores a smart pointer to a `Link` instance, decorated with methods that make it easier to use it. Since all copies of a given handle share the same link, they are all given access to the new pointee when any one of them is linked to a new object.

Outline of the `Handle` class template.

---

```

template <class Type>
class Handle {
    protected:
        class Link : public Observable, public Observer {
            public:
                explicit Link(const shared_ptr<Type>& h =
                                shared_ptr<Type>());
                void linkTo(const shared_ptr<Type>&);

                bool empty() const;
                void update() { notifyObservers(); }

            private:
                shared_ptr<Type> h_;
        };
        shared_ptr<Link<Type> > link_;
    public:
        explicit Handle(const shared_ptr<Type>& h =
                        shared_ptr<Type>());
        const shared_ptr<Type>& operator->() const;
        const shared_ptr<Type>& operator*() const;
        bool empty() const;
        operator shared_ptr<Observable>() const;
    };

template <class Type>
class RelinkableHandle : public Handle<Type> {
    public:
        explicit RelinkableHandle(const shared_ptr<Type>& h =
                                    shared_ptr<Type>());
        void linkTo(const shared_ptr<Type>&);

    };

```

---

The contained `shared_ptr<Link>` also gives the handle the means to be observed by other classes. The `Link` class is both an observer and an observable; it receives notifications from its pointee and forwards them to its own observers, as well as sending its own notification each time it is made to point to a different pointee. Handles take advantage of this behavior by defining an automatic conversion to `shared_ptr<Observable>` which simply returns the contained link. Thus, the statement

```
registerWith(h);
```

is legal and works as expected; the registered observer will receive notifications from both the link and (indirectly) the pointed object.

You might have noted that the means of relinking a handle (i.e., to have all its copies point to a different object) were not given to the `Handle` class itself, but to a derived `RelinkableHandle` class. The rationale for this is to provide control over which handle can be used for relinking—and especially over which handle can't. In the typical use case, a `Handle` instance will be instantiated (say, to store a yield curve) and passed to a number of instruments, pricing engines, or other objects that will store a copy of the handle and use it when needed. The point is that an object (or client code getting hold of the handle, if the object exposes it via an inspector) must not be allowed to relink the handle it stores, whatever the reason; doing so would affect a number of other object.<sup>20</sup>

The link should only be changed from the original handle—the main handle, if you like.

Given the frailty of human beings, we wanted this to be enforced by the compiler. Making the `linkTo` method a `const` one and returning `const` handles from our inspectors wouldn't work; client code could simply make a copy to obtain a non-`const` handle. Therefore, we removed `linkTo` from the `Handle` interface and added it to a derived class. The type system works nicely to our advantage. On the one hand, we can instantiate the main handle as a `RelinkableHandle` and pass it to any object expecting a `Handle`; automatic conversion from derived to base class will occur, leaving the object with a sliced but fully functional handle. On the other hand, when a copy of a `Handle` instance is returned from an inspector, there's no way to downcast it to `RelinkableHandle`.

## 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;
```

---

<sup>20</sup>This is not as far-fetched as it might seem; we've been bitten by it.

```

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

```

## Error reporting

There are a great many places in the library where some condition must be checked. Rather than doing it as

```

if (i >= v.size())
    throw Error("index out of range");

```

we wanted to express the intent more clearly, i.e., with a syntax like

```
require(i < v.size(), "index out of range");
```

where on the one hand, we write the condition to be satisfied and not its opposite; and on the other hand, terms such as `require`, `ensure`, or `assert`—which have a somewhat canonical meaning in programming—would tell whether we’re checking a precondition, a postcondition, or a programmer error.

We provided the desired syntax with macros. “Get behind thee,” I hear you say. True, macros have a bad name, and in fact they caused us a problem or two, as we’ll see below. But in this case, functions had a big disadvantage: they evaluate all their arguments. Many times, we want to create a moderately complex error message, such as

```
require(i < v.size(),
    "index " + to_string(i) + " out of range");
```

If `require` were a function, the message would be built whether or not the condition is satisfied, causing a performance hit that would not be acceptable. With a macro, the above is textually replaced by something like

```
if (!(i < v.size()))
    throw Error("index " + to_string(i) + " out of range");
```

which builds the message only if the condition is violated.

The next listing shows the current version of one of the macros, namely, `QL_REQUIRE`; the other macros are defined in a similar way.

Definition of the `QL_REQUIRE` macro.

---

```
#define QL_REQUIRE(condition,message) \
if (!(condition)) { \
    std::ostringstream _ql_msg_stream; \
    _ql_msg_stream << message; \
    throw QuantLib::Error(__FILE__,__LINE__, \
                         BOOST_CURRENT_FUNCTION, \
                         _ql_msg_stream.str()); \
} else
```

---

Its definition has a few more bells and whistles that might be expected. Firstly, we use an `ostringstream` to build the message string. This allows one to use a syntax like

```
QL_REQUIRE(i < v.size(),
    "index " << i << " out of range");
```

to build the message (you can see how that works by replacing the pieces in the macro body). Secondly, the `Error` instance is passed the name of the current function as well as the line and file where the error is thrown. Depending on a compilation flag, this information can be included in the error message to help developers; the default behavior is to not include it, since it's of little utility for users. Lastly, you might be wondering why we added an `else` at the end of the macro. That is due to a common macro pitfall, namely, its lack of a lexical scope. The `else` is needed by code such as

```
if (someCondition())
    QL_REQUIRE(i < v.size(), "index out of bounds");
else
    doSomethingElse();
```

Without the `else` in the macro, the above would not work as expected. Instead, the `else` in the code would pair with the `if` in the macro and the code would translate into

```
if (someCondition()) {
    if (!(i < v.size()))
        throw Error("index out of bounds");
    else
        doSomethingElse();
}
```

which has a different behavior.

As a final note, I have to describe a disadvantage of these macros. As they are now, they throw exceptions that can only return their contained message; no inspector is defined for any other relevant data. For instance, although an out-of-bounds message might include the passed index, no other method in the exception returns the index as an integer. Therefore, the information can be displayed to the user but would be unavailable to recovery code in catch clauses—unless one parses the message, that is; but that is hardly worth the effort. There's no planned solution at this time, so drop us a line if you have one.

## Disposable objects

The `Disposable` class template was an attempt to implement move semantics in C++03 code. To give credit where it's due, we took the idea and technique from an article by Andrei Alexandrescu ([Alexandrescu, 2003](#)) in which he described how to avoid copies when returning temporaries.

The basic idea is the one that was starting to float around in those years and that was given its final form in C++11: when passing a temporary object, copying it into another one is often less efficient than swapping its contents with those of the target. You want to move a temporary vector? Copy into the new object the pointer to its storage, instead of allocating a new one and copying the elements. In modern C++, the language itself supports move semantics with the concept of rvalue reference ([Hinnant et al, 2006](#)); the compiler knows when it's dealing with a temporary, and we can use `std::move` in the few cases when we want to turn an object into one. In our implementation, shown in the following listing, we don't have such support; you'll see the consequences of this in a minute.

Implementation of the `Disposable` class template.

---

```
template <class T>
class Disposable : public T {
public:
    Disposable(T& t) {
        this->swap(t);
    }
    Disposable(const Disposable<T>& t) : T() {
        this->swap(const_cast<Disposable<T>&>(t));
    }
    Disposable<T>& operator=(const Disposable<T>& t) {
        this->swap(const_cast<Disposable<T>&>(t));
        return *this;
    }
};
```

---

The class itself is not much to look at. It relies on the template argument implementing a `swap` method; this is where any resource contained inside the class are swapped (hopefully in a cheap way) instead of copied. The constructors and the assignment operator all use this to move stuff around without copies—with a difference, depending on what is passed. When building a `Disposable` from another one, we take it by `const` reference because we want the argument to bind to temporaries; that's what most disposables will be. This forces us to use a `const_cast` in the body, when it's time to call `swap` and take the resources from the disposable. When building a `Disposable` from a non-disposable object, instead, we take it as a non-`const` reference; this is to prevent ourselves from triggering unwanted destructive conversions and from finding ourselves with the empty husk of an object when we thought to have a usable one. This, however, has a disadvantage; I'll get to it in a minute.

The next listing shows how to retrofit `Disposable` to a class; `Array`, in this case.

Use of the `Disposable` class template in the `Array` class.

---

```
Array::Array(const Disposable<Array>& from)
: data_((Real*)(0)), n_(0) {
    swap(const_cast<Disposable<Array>&>(from));
}

Array& Array::operator=(const Disposable<Array>& from) {
    swap(const_cast<Disposable<Array>&>(from));
    return *this;
}

void Array::swap(Array& from) {
```

```

    data_.swap(from.data_);
    std::swap(n_, from.n_);
}

```

---

As you see, we need to add a constructor and an assignment operator taking a `Disposable` (in C++11, they would be a move constructor and a move assignment operators taking an rvalue reference) as well as the `swap` method that will be used in all of them. Again, the constructors take the `Disposable` by `const` reference and cast it later, in order to bind to temporaries—although now that I think of it, they could take it by copy, adding another cheap swap.

Finally, the way `Disposable` is used is by returning it from function, like in the following code:

```

Disposable<Array> ones(Size n) {
    Array result(n, 1.0);
    return result;
}

Array a = ones(10);

```

Returning the array causes it to be converted to `Disposable`, and assigning the returned object causes its contents to be swapped into `a`.

Now, you might remember that I talked about a disadvantage when I showed you the `Disposable` constructor being safe and taking an object by non-`const` reference. It's that it can't bind to temporaries; therefore, the function above can't be written more simply as:

```

Disposable<Array> ones(Size n) {
    return Array(n, 1.0);
}

```

because that wouldn't compile. This forces us to take the more verbose route and give the array a name.<sup>21</sup>

Nowadays, of course, we'd use rvalue references and move constructors and forget all about the above. To tell the truth, I've a nagging suspicion that `Disposable` might be getting in the way of the compiler and doing more harm than good. Do you know the best way to write code like the above and avoid abstraction penalty in modern C++? It's this one:

---

<sup>21</sup>Well, it doesn't actually *force* us, but writing `return Disposable<Array>(Array(n, 10))` is even uglier than the alternative.

```
Array ones(Size n) {
    return Array(n, 1.0);
}

Array a = ones(10);
```

In C++17, the copies that might have been done when returning the array and when assigning it are guaranteed to be elided (that is, the compiler will generate code that builds the returned array directly inside the one we're assigning); most recent compilers have been doing that for a while, without waiting for the standard to bind them. It's called RVO, for Return Value Optimization, and using `Disposable` prevents it and thus might make the code slower instead of faster.

## Design patterns

A few design patterns were implemented in QuantLib. You can refer to the Gang of Four book ([Gamma et al, 1995](#)) for a description of such patterns; so why do I write about them? Well, as once noted by G. K. Chesterton,

[p]oets have been mysteriously silent on the subject of cheese

and the Gang was just as silent on a number of issues that come up when you write actual implementations—through no fault of them, mind you. The variations are almost limitless, and they were only four.

Thus, I will use this final section to point out a few ways in which our implementations were tailored to the requirements of the library.

### The Observer pattern

The use of the Observer pattern in the QuantLib library is widespread; you've seen it used in [chapter 2](#) and [chapter 3](#) to let financial instruments and term structures keep track of changes and recalculate when needed.

Our version of the pattern (sketched in the next listing) is close enough to that described in the Gang of Four book; but as I mentioned, there are questions and problems that weren't discussed there.

Sketch of the **Observable** and **Observer** classes.

---

```

class Observable {
    friend class Observer;
public:
    void notifyObservers() {
        for (iterator i=observers_.begin();
              i!=observers_.end(); ++i) {
            try {
                (*i)->update();
            } catch (std::exception& e) {
                // store information for later
            }
        }
    }
private:
    void registerObserver(Observer* o) {
        observers_.insert(o);
    }
    void unregisterObserver(Observer* );
    list<Observer*> observers_;
};

class Observer {
public:
    virtual ~Observer() {
        for (iterator i=observables_.begin();
              i!=observables_.end(); ++i)
            (*i)->unregisterObserver(this);
    }
    void registerWith(const shared_ptr<Observable>& o) {
        o->registerObserver(this);
        observables_.insert(o);
    }
    void unregisterWith(const shared_ptr<Observable>& );
    virtual void update() = 0;
private:
    list<shared_ptr<Observable> > observables_;
};

```

---

For instance: what information should we include in the notification? In our implementation, we went for minimalism—all that an observer gets to know is that something changed. It would have been possible to provide more information (e.g., by having the `update` method take the notifying

observable as an argument) so that observers could select what to recalculate and save a few cycles; but I don't think that this feature was worth the added complexity.

Another question: what happens if an observer raises an exception from its update method? This would happen when an observable is sending a notification, i.e., while the observable is iterating over its observers, calling update on each one. If the exception were to propagate, the loop would be aborted and a number of observers would not receive the notification—bad. Our solution was to catch any such exception, complete the loop, and raise an exception at the end if anything went wrong. This causes the original exceptions to be lost, which is not good either; however, we felt this to be the lesser of the two evils.

Onwards to the third issue: that is, copy behavior. It is not very clear what should happen when an observer or an observable are copied. Currently, what seemed a sensible choice is implemented: on the one hand, copying an observable results in the copy not having any observer; on the other hand, copying an observer results in the copy being registered with the same observables as the original. However, other behaviors might be considered; as a matter of fact, the right choice might be to inhibit copying altogether.

The big problems, however, were two. First: we obviously had to make sure that the lifetimes of the observer and observables were managed properly, meaning that no notification should be sent to an already deleted object. To do so, we had observers store shared pointers to their observables, which ensures that no observable is deleted before an observer is done with it. The observers will unregister with any observable before being deleted, which in turn makes it safe for observables to store a list of raw pointers to their observers.

This, however, is only guaranteed to work in a single-threaded setting; and we are exporting QuantLib bindings to C# and Java, where unfortunately there is always another thread where the garbage collector is busy deleting stuff. Every once in a while, this caused random crashes as a notification was sent to a half-deleted object. Once the problem was understood, it was fixed (hi, Klaus); however, the fix slows down the code, so it's inactive by default and can be enabled by a compilation switch. Use it if you need the C# or Java bindings.

The second big problem is<sup>22</sup> that, like in the Jerry Lee Lewis' song, there's a whole lotta notifyin' going on. A change of date can easily trigger tens or hundreds of notifications; and even if most of the update methods only set a flag and forward the call, the time adds up.

People using QuantLib in applications where calculation time is paramount, such as CVA/XVA (hi, Peter) have worked around the problem by disabling notifications and recalculating explicitly. A step towards reducing notification time would be to remove the middlemen, and shorten the chains of notifications; however, this is not possible due to the ubiquitous presence of the Handle class in the chains. Handles can be relinked, and thus chains of dependencies can change even after objects are built.

In short, the problem is still not solved. You know where to find us if you have any bright ideas.

---

<sup>22</sup>Notice that I didn't say *was*.

## The Singleton pattern

The Gang of Four devoted the first part of their book to creational patterns. While logically sound, this choice turned out to have an unfortunate side effect: all too often, overzealous programmers would start to read the book and duly proceed to sprinkle their code with abstract factories and singletons. Needless to say, this does less than intended for the clarity of the code.

You might suspect the same reason for the presence of a `Singleton` class template in QuantLib. (Quite maliciously, I might add. Shame on you.) Fortunately, we can base our defense on version-control logs; such class was added to the library later than, say, Observer (a behavioral pattern) or Composite (a structural one).

Our default implementation is shown in the following listing.

Interface of the `Singleton` class template.

---

```
template <class T>
class Singleton : private noncopyable {
public:
    static T& instance();
protected:
    Singleton();
};

#if defined(QL_ENABLE_SESSIONS)
// the definition must be provided by the user
Integer sessionId();
#endif

template <class T>
T& Singleton<T>::instance() {
    static map<Integer, shared_ptr<T> > instances_;
    #if defined(QL_ENABLE_SESSIONS)
    Integer id = sessionId();
    #else
    Integer id = 0;
    #endif
    shared_ptr<T>& instance = instances_[id];
    if (!instance)
        instance = shared_ptr<T>(new T);
    return *instance;
}
```

---

It's based on the Curiously Recurring Template Pattern, that I described in [chapter 7](#); to be a singleton, a class `C` needs to inherit from `Singleton<C>`, to provide a private constructor taking on arguments,

and to make `Singleton<C>` a friend so that it can use it. You can see an example in [the Settings class](#).

As suggested by Scott Meyers ([Meyers, 2005](#)), the map holding the instances (bear with me) is defined as a static variable inside the `instance` method. This prevents the so-called static initialization order fiasco, in which the variable is used before being defined, and in C++11 it has the additional guarantee that the initialization is thread-safe (even though that's not the whole story, as we'll see).

Now, you might have a few questions; e.g., why a map of instances if this is supposed to be a singleton? Well, that's because having a single instance might be limiting; for instance—no pun intended—you might want to perform simultaneous calculations on different evaluation dates. Thus, we tried to mitigate the problem by allowing one `Singleton` instance per thread. This is enabled by a compilation flag, and causes the `instance` method to use the `#if` branch in which it gets an id from a `sessionId` function and uses it to index into the map. If you enable per-thread singletons, you must also provide the latter function; it will probably be something like

```
Integer sessionId() {
    return /* `some unique thread id from your system API` */ ;
}
```

in which you will identify the thread using the functions made available by your operating system (or some threading library), turn the identifier into a unique integer, and return it. In turn, this will cause the `instance` method to return a unique instance per each thread. If you don't enable the feature, instead, the id will always ever be `0` and you'll always get the same instance. In this case, you probably don't want to use threads at all—and in the other case, you obviously do, but you have to be careful anyway: see the discussion in [the section on global settings](#).

You might also be asking yourself why I said that this is our *default* implementation. Nice catch. There are others, which I won't show here and which are enabled by a number of compilation flags. On the one hand, it turned out that the static-variable implementation didn't work when compiled as managed C++ code under .NET (at least with older Visual Studio compilers), so in that case we switch it with one in which the map is a static class variable. On the other hand, if you want to use a global `Singleton` instance in a multi-threaded setting, you want to make sure that the initialization of the `Singleton` instance is thread-safe (I'm talking about the instance itself, not the map containing it; it's where the new `T` is executed). This requires locks, mutexes and stuff, and we don't want to go near any of that in the default single-threaded setting; therefore, that code is behind yet another compilation flag. You can look at it in the library, if you're interested.

Your last question might be whether we should have a `Singleton` class at all—and it's a tough one. Again, I refer you to the previous discussion of global settings. At this time, much like democracy according to Winston Churchill, it seems to be the worst solution except for all the others.

## The Visitor pattern

Our implementation, shown in the next listing, follows the Acyclic Visitor pattern ([Martin, 1997](#)) rather than the one in the Gang of Four book: we defined a degenerate `AcyclicVisitor` class, to be

used in the interfaces, and a class template `Visitor` which defines the pure virtual `visit` method for its template argument.

Interface of the `AcylicVisitor` class and of the `Visitor` class template.

---

```
class AcyclicVisitor {
public:
    virtual ~AcyclicVisitor() {}

};

template <class T>
class Visitor {
public:
    virtual ~Visitor() {}
    virtual void visit(T&) = 0;
};
```

---

The pattern also needs support from any class hierarchy that we want to be visitable; an example is shown in the listing that follows.

Implementation of the Visitor pattern in a class hierarchy.

---

```
void Event::accept(AcyclicVisitor& v) {
    Visitor<Event>* v1 = dynamic_cast<Visitor<Event>>(&v);
    if (v1 != 0)
        v1->visit(*this);
    else
        QL_FAIL("not an event visitor");
}

void CashFlow::accept(AcyclicVisitor& v) {
    Visitor<CashFlow>* v1 =
        dynamic_cast<Visitor<CashFlow>>(&v);
    if (v1 != 0)
        v1->visit(*this);
    else
        Event::accept(v);
}

void Coupon::accept(AcyclicVisitor& v) {
    Visitor<Coupon>* v1 = dynamic_cast<Visitor<Coupon>>(&v);
    if (v1 != 0)
        v1->visit(*this);
    else
```

---

```
CashFlow::accept(v);  
}
```

---

Each of the classes in the hierarchy (or at least, those that we want to be specifically visitable) need to define an `accept` method that takes a reference to `AcyclicVisitor`. Each of the methods tries to cast the passed visitor to the specific `Visitor` instantiation for the corresponding class. A successful cast means that the visitor defines a `visit` method taking this specific class, so we invoke it. A failed cast means that we have to look for an fallback. If the class is not the root of the hierarchy (like `CashFlow` or `Coupon` in the listing) we can call the base-class implementation of `accept`, which in turn will try the cast. If we're at the root, like the `Event` class, we have no further fallback and we raise an exception.<sup>23</sup>

Finally, a visitor is implemented as the `BPSCalculator` class in [chapter 4](#). It inherits from `AcyclicVisitor`, so that it can be passed to the various `accept` methods, as well as from an instantiation of the `Visitor` template for each class for which it will provide a `visit` method. It will be passed to the `accept` method of some instance, which will eventually call one of the `visit` methods or raise an exception.

I already discussed the usefulness of the Visitor pattern in [chapter 4](#), so I refer you to it (the unsurprising summary: it depends). Therefore, I will only spend a couple of words on why we chose Acyclic Visitor.

In short, the Gang-of-Four version of Visitor might be a bit faster, but it's a lot more intrusive; in particular, every time you add a new class to the visitable hierarchy you're also forced to go and add the corresponding `visit` method to each existing visitor (where by "forced" I mean that your code wouldn't compile if you didn't). With Acyclic Visitor, you don't need to do it; the `accept` method in your new class will fail the cast and fallback to its base class.<sup>24</sup> Mind you, this is convenient but not necessarily a good thing (like a lot of things in life, I might add): you should review existing visitors anyway, check whether the fallback implementation makes sense for your class, and add a specific one if it doesn't. But I think that the disadvantage of not having the compiler warn you is more than balanced by the advantage of not having to write a `visit` method for each cash-flow class when, as in `BPSCalculator`, a couple will suffice.

<sup>23</sup>Another alternative would be to do nothing, but we preferred not to fail silently.

<sup>24</sup>In fact, you're not even required to define an `accept` method; you could just inherit it. However, this would prevent visitors to target this specific class.

## B. Code conventions

Every programmer team has a number of conventions to be used while writing code. Whatever the conventions (several exist, which provides a convenient *casus belli* for countless wars) adhering to them helps all developers working on the same project,<sup>25</sup> as they make it easier to understand the code; a reader familiar with their use can distinguish at a glance between a macro and a function, or between a variable and a type name.

The following listing briefly illustrates the conventions used throughout the QuantLib library. Following the advice in [Sutter and Alexandrescu, 2004](#), we tried to reduce their number at a minimum, enforcing only those conventions which enhance readability.

Illustration of QuantLib code conventions.

---

```
#define SOME_MACRO

typedef double SomeType;

class SomeClass {
public:
    typedef Real* iterator;
    typedef const Real* const_iterator;
};

class AnotherClass {
public:
    void method();
    Real anotherMethod(Real x, Real y) const;
    Real member() const; // getter, no "get"
    void setMember(Real); // setter
private:
    Real member_;
    Integer anotherMember_;
};

struct SomeStruct {
    Real foo;
    Integer bar;
};
```

---

<sup>25</sup>However, the QuantLib developers are human. As such, they sometimes fail to follow the rules I am describing.

```
Size someFunction(Real parameter,
                  Real anotherParameter) {
    Real localVariable = 0.0;
    if (condition) {
        localVariable += 3.14159;
    } else {
        localVariable -= 2.71828;
    }
    return 42;
}
```

---

Macros are in all uppercase, with words separated by underscores. Type names start with a capital and are in the so-called camel case; words are joined together and the first letter of each word is capitalized. This applies to both type declarations such as `SomeType` and class names such as `SomeClass` and `AnotherClass`. However, an exception is made for type declarations that mimic those found in the C++ standard library; this can be seen in the declaration of the two iterator types in `SomeClass`. The same exception might be made for inner classes.

About everything else (variables, function and method names, and parameters) are in camel case and start with a lowercase character. Data members of a class follow the same convention, but are given a trailing underscore; this makes it easier to distinguish them from local variables in the body of a method (an exception is often made for public data members, especially in structs or struct-like classes). Among methods, a further convention is used for getters and setters. Setter names are created by adding a leading `set` to the member name and removing the trailing underscore. Getter names equal the name of the returned data member without the trailing underscore; no leading `get` is added. These conventions are exemplified in `AnotherClass` and `SomeStruct`.

A much less strict convention is that the opening brace after a function declaration or after an `if`, `else`, `for`, `while`, or `do` keyword are on the same line as the preceding declaration or keyword; this is shown in `someFunction`. Moreover, `else` keywords are on the same line as the preceding closing brace; the same applies to the `while` ending a `do` statement. However, this is more a matter of taste than of readability; therefore, developers are free to use their own conventions if they cannot stand this one. The shown function exemplifies another convention aimed at improving readability, namely, that function and method arguments should be aligned vertically if they do not fit a single line.

# QuantLib license

QuantLib is

- © 2000, 2001, 2002, 2003 RiskMap srl
- © 2001, 2002, 2003 Nicolas Di Césaré
- © 2001, 2002, 2003 Sadruddin Rejeb
- © 2002, 2003, 2004 Decillion Pty(Ltd)
- © 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2014, 2015 Ferdinando Ametrano
- © 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2014, 2016, 2017, 2018, 2019 StatPro Italia srl
- © 2003, 2004, 2007 Neil Firth
- © 2003, 2004 Roman Gitlin
- © 2003 Niels Elken Sønderby
- © 2003 Kawanishi Tomoya
- © 2004 FIMAT Group
- © 2004 M-Dimension Consulting Inc.
- © 2004 Mike Parker
- © 2004 Walter Penschke
- © 2004 Gianni Piolanti
- © 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019 Klaus Spanderen
- © 2004 Jeff Yu
- © 2005, 2006, 2008 Toyin Akin
- © 2005 Sercan Atalik
- © 2005, 2006 Theo Boafo
- © 2005, 2006, 2007, 2009 Piter Dias
- © 2005, 2013 Gary Kennedy
- © 2005, 2006, 2007 Joseph Wang
- © 2005 Charles Whitmore
- © 2006, 2007 Banca Profilo S.p.A.
- © 2006, 2007 Marco Bianchetti
- © 2006 Yiping Chen
- © 2006 Warren Chou
- © 2006, 2007 Cristina Duminuco
- © 2006, 2007 Giorgio Facchinetti
- © 2006, 2007 Chiara Fornarola

- © 2006 Silvia Frasson
- © 2006 Richard Gould
- © 2006, 2007, 2008, 2009, 2010 Mark Joshi
- © 2006, 2007, 2008 Allen Kuo
- © 2006, 2007, 2008, 2009, 2012 Roland Lichters
- © 2006, 2007 Katiuscia Manzoni
- © 2006, 2007 Mario Pucci
- © 2006, 2007 François du Vignaud
- © 2007 Affine Group Limited
- © 2007 Richard Gomes
- © 2007, 2008 Laurent Hoffmann
- © 2007, 2008, 2009, 2010, 2011 Chris Kenyon
- © 2007 Gang Liang
- © 2008, 2009, 2014, 2015, 2016 Jose Aparicio
- © 2008 Yee Man Chan
- © 2008, 2011 Charles Chongseok Hyun
- © 2008 Piero Del Boca
- © 2008 Paul Farrington
- © 2008 Lorella Fatone
- © 2008, 2009 Andreas Gaida
- © 2008 Marek Glowacki
- © 2008 Florent Grenier
- © 2008 Frank Hövermann
- © 2008 Simon Ibbotson
- © 2008 John Maiden
- © 2008 Francesca Mariani
- © 2008, 2009, 2010, 2011, 2012, 2014 Master IMAFA - Polytech'Nice Sophia - Université de Nice Sophia Antipolis
- © 2008, 2009 Andrea Odetti
- © 2008 J. Erik Radmall
- © 2008 Maria Cristina Recchioni
- © 2008, 2009, 2012, 2014 Ralph Schreyer
- © 2008 Roland Stamm
- © 2008 Francesco Zirilli
- © 2009 Nathan Abbott
- © 2009 Sylvain Bertrand
- © 2009 Frédéric Degraeve
- © 2009 Dirk Eddelbuettel
- © 2009 Bernd Engelmann
- © 2009, 2010, 2012 Liquidnet Holdings, Inc.
- © 2009 Bojan Nikolic
- © 2009, 2010 Dimitri Reiswich

- © 2009 Sun Xiuxin
- © 2010 Kakhkhор Abdijalilov
- © 2010 Hachemi Benyahia
- © 2010 Manas Bhatt
- © 2010 DeriveXperts SAS
- © 2010, 2014 Cavit Hafizoglu
- © 2010 Michael Heckl
- © 2010 Slava Mazur
- © 2010, 2011, 2012, 2013 Andre Miemiec
- © 2010 Adrian O' Neill
- © 2010 Robert Philipp
- © 2010 Alessandro Roveda
- © 2010 SunTrust Bank
- © 2011, 2013, 2014 Fabien Le Floc'h
- © 2012, 2013 Grzegorz Andruszkiewicz
- © 2012, 2013, 2014, 2015, 2016, 2017, 2018 Peter Caspers
- © 2012 Mateusz Kapturski
- © 2012 Simon Shakeshaft
- © 2012 Édouard Tallent
- © 2012 Samuel Tebege
- © 2013 BGC Partners L.P.
- © 2013, 2014 Cheng Li
- © 2013 Yue Tian
- © 2014, 2017 Francois Botha
- © 2014, 2015 Johannes Goettker-Schnetmann
- © 2014 Michal Kaut
- © 2014, 2015 Bernd Lewerenz
- © 2014, 2015, 2016 Paolo Mazzocchi
- © 2014, 2015 Thema Consulting SA
- © 2014, 2015, 2016 Michael von den Driesch
- © 2015 Riccardo Barone
- © 2015 CompatibL
- © 2015, 2016 Andres Hernandez
- © 2015 Dmitri Nesteruk
- © 2015 Maddalena Zanzi
- © 2016 Nicholas Bertocchi
- © 2016 Stefano Fondi
- © 2016, 2017 Fabrice Lecuyer
- © 2016, 2019 Eisuke Tani
- © 2017 BN Algorithms Ltd
- © 2017 Paul Giltinan
- © 2017 Werner Kuerzinger

- © 2017 Oleg Kulkov
- © 2017 Joseph Jeisman
- © 2018 Tom Anderson
- © 2018 Alexey Indiryakov
- © 2018 Jose Garcia
- © 2018 Matthias Groncki
- © 2018 Matthias Lungwitz
- © 2018 Sebastian Schlenkrich
- © 2018 Roy Zywina
- © 2019 Aprexo Limited
- © 2019 Wojciech Slusarski

QuantLib includes code taken from Peter Jäckel's book "Monte Carlo Methods in Finance".

QuantLib includes software developed by the University of Chicago, as Operator of Argonne National Laboratory.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the names of the copyright holders nor the names of the QuantLib Group and its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided by the copyright holders and contributors "as is" and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the copyright holders or contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

# Bibliography

- D. Abrahams, *Want Speed? Pass by Value*. In *C++ Next*, 2009.
- D. Adams, *So Long, and Thanks for all the Fish*. 1984.
- A. Alexandrescu, *Move Constructors*. In *C/C++ Users Journal*, February 2003.
- F. Ametrano and M. Bianchetti, *Everything You Always Wanted to Know About Multiple Interest Rate Curve Bootstrapping but Were Afraid to Ask*. SSRN working papers series n.2219548, 2013.
- J. Barton and L.R. Nackman, *Dimensional Analysis*. In *C++ Report*, January 1995.
- T. Becker, *On the Tension Between Object-Oriented and Generic Programming in C++*. 2007.
- Boost C++ libraries. <http://boost.org>.
- M. K. Bowen and R. Smith, Derivative formulae and errors for non-uniformly spaced points. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 461, pages 1975–1997. The Royal Society, 2005.
- D. Brigo and F. Mercurio, *Interest Rate Models — Theory and Practice*, 2nd edition. Springer, 2006.
- Mel Brooks (director), *Young Frankenstein*. Twentieth Century Fox, 1974.
- W.E. Brown, *Toward Opaque Typedefs for C++1Y, v2*. C++ Standards Committee Paper N3741, 2013.
- L. Carroll, *The Hunting of the Snark*. 1876.
- G.K. Chesterton, *Alarms and Discursions*. 1910.
- M.P. Cline, G. Lomow and M. Girou, *C++ FAQs*, 2nd edition. Addison-Wesley, 1998.
- J.O. Coplien, *A Curiously Recurring Template Pattern*. In S.B. Lippman, editor, *C++ Gems*. Cambridge University Press, 1996.
- C.S.L. de Graaf. *Finite Difference Methods in Derivatives Pricing under Stochastic Volatility Models*. Master's thesis, Mathematisch Instituut, Universiteit Leiden, 2012.
- C. Dickens, *Great Expectations*. 1860.
- P. Dimov, H.E. Hinnant and D. Abrahams, *The Forwarding Problem: Arguments*. C++ Standards Committee Paper N1385, 2002.
- M. Dindal (director), *The Emperor's New Groove*. Walt Disney Pictures, 2000.
- D.J. Duffy, *Finite Difference Methods in Financial Engineering: A Partial Differential Equation Approach*. John Wiley and Sons, 2006.
- M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd edition.

- Addison-Wesley, 2003.
- M. Fowler, *Fluent Interface*. 2005.
- M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Element of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- P. Glasserman, *Monte Carlo Methods in Financial Engineering*. Springer, 2003.
- D. Gregor, *A Brief Introduction to Variadic Templates*. C++ Standards Committee Paper N2087, 2006.
- H.E. Hinnant, B. Stroustrup and B. Kozicki, *A Brief Introduction to Rvalue References*. C++ Standards Committee Paper N2027, 2006.
- A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.
- International Standards Organization, *Programming Languages — C++*. International Standard ISO/IEC 14882:2014.
- International Swaps and Derivatives Associations, *Financial products Markup Language*.
- P. Jäckel, *Monte Carlo Methods in Finance*. John Wiley and Sons, 2002.
- J. Kerievsky, *Refactoring to Patterns*. Addison-Wesley, 2004.
- R. Kleiser (director), *Grease*. Paramount Pictures, 1978.
- H.P. Lovecraft, *The Call of Cthulhu*. 1928.
- R.C. Martin, *Acyclic Visitor*. In *Pattern Languages of Program Design 3*. Addison-Wesley, 1997.
- S. Meyers, *Effective C++*, 3rd edition. Addison-Wesley, 2005.
- N.C. Myers, *Traits: a new and useful template technique*. In The C++ Report, June 1995.
- G. Orwell, *Animal Farm*. 1945.
- W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, *Numerical Recipes in C*, 2nd edition. Cambridge University Press, 1992.
- QuantLib. <http://quantlib.org>.
- E. Queen, *The Roman Hat Mystery*. 1929.
- V. Simonis and R. Weiss, *Exploring Template Template Parameters*. In *Perspectives of System Informatics*, number 2244 in Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2001.
- B. Stroustrup, *The C++ Programming Language*, 4th edition. Addison-Wesley, 2013.
- H. Sutter, *You don't know const and mutable*. In Sutter's Mill, 2013.
- H. Sutter and A. Alexandrescu, *C++ Coding Standards*. Addison-Wesley, 2004.
- T. Veldhuizen, *Techniques for Scientific C++*. Indiana University Computer Science Technical Report

TR542, 2000.

H.G. Wells, *The Shape of Things to Come*. 1933.

P.G. Wodehouse, *My Man Jeeves*. 1919.