Implementing QuantLib

Luigi Ballabio

A Odds and ends

Of shoes – and ships – and sealing-wax –
Of cabbages – and kings –

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,[1]

> [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;[2] I point those who desire such documentation to the QuantLib reference manual, available at the QuantLib web site [19].

A.1. 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 to make dimensional analysis to those who, like me, were used to it in a previous life as a physicist; seemingly harmless expressions such as `exp(r)` or `r+s*t` can be quickly flagged as fishy if they are preceded by `Rate r`, `Spread s`, and `Time t`.

[1] No, it's not a mistake. It is an inaccurately-named trilogy of five books. It's a long story.
[2] Nor automatic, not hydromatic. That would be the Grease Lightning.

A.2. 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.

A.2.1. Dates and periods

The `Date` class models the basic concept of date. A `Date` instance represents a specific day such as October 23rd, 2007—today's date as I write. 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 is documented in the reference manual.

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 actual 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.

A.2.2. 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 number of business days.

Listing A.1: Outline of the `Calendar` class.

```
class Calendar {
  protected:
    class Impl {
      public:
        virtual ~Impl() {}
        virtual bool isBusinessDay(const Date&) const = 0;
    };
    boost::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
};
```


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 need to be passed and stored in `shared_ptrs`. The class was simple enough on a conceptual level and was used frequently enough that we wanted users to instantiate it and pass it around more easily, namely, without the added verbosity of dynamic allocation.

The final solution was the one shown in listing A.1. It is a variation of the pimpl idiom, also reminiscent of the Strategy or Bridge patterns: `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`.

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.

A.2.3. 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.)

A.2.4. Schedules

The `Schedule` class, shown in listing A.2, is used to generate sequences of coupon dates. Following practice and ISDA conventions, it has to accept a lot of parameters; you can see them as the argument list of its constructor. They're probably too many, which is why the library uses the Named Parameter Idiom (already described in

Listing A.2: Interface of the `Schedule` class.

```
class Schedule {
  public:
    Schedule();
    Schedule(const Date& effectiveDate,
             const Date& termination Date,
             const Period& tenor,
             const Calendar& calendar,
             BusinessDayConvention convention,
             BusinessDayConvention terminationDateConvention,
             DateGeneration::Rule rule,
             bool endOfMonth,
             const Date& firstDate = Date(),
             const Date& nextToLastDate = Date());

    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;

    Date previousDate(const Date& refDate) const;
    Date nextDate(const Date& refDate) const;
    bool isRegular(Size i) const;
    const Calendar& calendar() const;
    const Date& startDate() const;
    ... // other inspectors
};
```


section 4.2.5) 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`.

A.3. 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 on 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 might 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.

Listing A.3 shows the current version of one of the macros, namely, `QL_REQUIRE`; the other macros are defined in a similar way. Its definition has a few more bells

Listing A.3: 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
```

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 datum 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.

A.4. 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.

A.4.1. 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. The second resulted in the `Quote` class, shown in listing A.4.

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 section 3.2.3) 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 an genuine source of external values; that would be the `SimpleQuote` class, shown in listing A.5. 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`

Listing A.4: Interface of the `Quote` class.

```
class Quote : public virtual Observable {
   public:
     virtual ~Quote() {}
     virtual Real value() const = 0;
     virtual bool isValid() const = 0;
 };
```

Listing A.5: 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_;
};
```


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.[3]

I'll conclude this section with two short notes. The first is that, although nobody provided implementations yet, the 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 them in a uniform way. The second is that the type of the quoted values is constrained to `Real`. However, this has not been a limitation so far (and most likely, it won't be until the `Money` class is extensively used.) Besides, it's now too late to define `Quote` as a class template.

[3] 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.

Listing A.6: 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
};
```

A.4.2. Interest rates

The `InterestRate` class (shown in listing A.6) 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.

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.[4]

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

```
enum Compounding { Simple,
                   Compounded(Frequency),
                   Continuous,
                   SimpleThenCompounded(Frequency)
};
```

which would be similar to algebraic data types in functional languages, or case

[4] 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."

Listing A.7: 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();
};
```


classes in Scala;[5] 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.

A.4.3. 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 listing A.7, all its methods have to do with index fixings. 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; the `addFixing` method stores an index fixing (or many, in other overloads not shown here;) the `fixing` method retrieves a fixing for a past date or forecasts one for a future date; the `clearFixing` method clears all stored fixings for the given index; and even the `name` method, which returns an identifier that must be unique for each index class, is used to index (pun not intended) into a map of stored fixings.

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

[5] Both support pattern matching on an object, which is like a neater `switch` on steroids. Go have a look when you have some time.

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, 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.) In any case, this is one of the things we'll have to rethink in the next big QuantLib revision,[6] in which we'll have to tackle concurrency.

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.) Moving the inheritance from `Observer` down the hierarchy was not a design choice, but rather an artifact of the evolution of the code that might change in future releases. However, even if we were to inherit `Index` from `Observer` instead, we would still be forced to have some code duplication in derived classes, for a reason which is probably worth describing in more detail.

When an instrument is registered with an index, updates can happen for two reasons. One is that the index depends on other observables to forecast its fixings, registered with them (this is done in each derived class, as each class has different observables) and they send out a notification. In this case, the index simply forwards notification to its own observers.

The other reason is that a new fixing might be made available, and that's more tricky to handle. The fixing is usually stored by a call to `addFixing`; but we can't just call the `notifyObservers` method from there. The reason is that we want the fixings to be shared; if we store today's 3-months Euribor fixing, we want all instruments and term structures that depend on it to be notified, and not just those which happen to have registered with the particular instance on which we called `addFixing`. However, instruments and curves register with `Index` instances, so that's where notifications must come from.

The solution is to have all instances of the same index[7] 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, it provides the means to register and receive notification when one or more fixings are added for a specific tag (it does this by making the sets instances of

[6] If these past years are any indication, we might expect it around 2020. No, just kidding. Maybe.

[7] 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.

the `ObservableValue` class, described later in this appendix. You don't need the details here, but you can gather them from the source if you want.)

All pieces are now in place. Upon construction, all `Index` instances will ask `IndexManager` for the shared observable corresponding to the tag returned by their `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.

And this is where C++ 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.[8] From here stems the code duplication I mentioned earlier; in order to work, the above method call must be added to the constructor of each derived index class.

As an example of a concrete class derived from `Index`, listing A.8 sketches the `InterestRateIndex` class. As you might expect, such class defines a good deal of specific behavior besides that inherited 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; after which, the index registers with a couple of observables. The first is the global evaluation date; this is needed since, as I'll explain in a minute or two, there's a bit of date-specific behavior in the class that is triggered when an instance is asked for today's fixing; if today becomes yesterday, the returned values might change. The second observable is the one returned from `IndexManager` and providing notifications when new fixings are stored. This is possible because the `InterestRateIndex` has the information needed to implement a `name` method. 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 (see the footnote on this page,) they would register with the wrong notifier. Unfortunately, this can't be enforced in C++, which doesn't have a keyword

[8] 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 constructor body.

Listing A.8: 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,
                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_;
};
```

Listing A.8 (continued.)

```
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);
}
```

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 and probably more error-prone.

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, and 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. If today's fixing was requested, the index first tries looking in the `IndexManager`; if the fixing was stored, the method returns it, but no exception is raised otherwise since it just means that 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 index must know when today's date changes, since its behavior depends on it.

Finally, the `InterestRateIndex` class defines other methods that are not inherited. Most of them are inspectors that return stored data such as the family name or

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 might make 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 couldn't fit it since we have only one tenor in the base class. 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 most likely not one such class.

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.

A.4.4. Exercises and payoffs

A.5. Math-related classes

A.5.1. Interpolations

A.5.2. One-dimensional solvers

A.6. Global settings

The `Settings` class is a singleton holding information global to the whole library. Such information includes the evaluation date, defaulting to today's date and used for the pricing of instruments and the fixing of any other quantity.

An aspect of this class (outlined in listing A.9) is worth observing. 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 listing A.10) 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.

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 like just a `Date` instance, as in:

```
Date d2 =
    calendar.adjust(Settings::instance().evaluationDate());
```

Listing A.9: 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;
    ...
};
```

Listing A.10: 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 boost::shared_ptr<Observable>() const {
      return observable_;
    }
  private:
    T value_;
    boost::shared_ptr<Observable> observable_;
};
```

which triggers an automatic conversion; and on the gripping hand, although a simple assignment syntax can be used for setting the evaluation date as in:

```
Settings::instance().evaluationDate() = d;
```

such statement will cause all observers to be notified of the date change.

Finally, the `Settings` class provides some support for thread-local settings. This will be discussed in a later section, in the context of the `Singleton` class.

A.7. Utility classes

A number of classes exist and are used in QuantLib which do not 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 such classes.

A.7.1. 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; suffice to say that they are difficult ones already when all goes well, but get even worse when the possibility of exceptions enter the picture. The difficulty of the task is enough to discourage manual management; therefore, some way must be found to automate the process.

The weapons of choice in the C++ developers 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. The ones we chose are the smart pointers from the Boost libraries [2] (most notably `shared_ptr`) that will be part of the next revision of the ANSI/ISO C++ standard. I refer the reader to the Boost site for documentation, and only mention here 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 on the next page 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 writing

```
boost::shared_ptr<boost::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; and on the gripping hand,

Aside: pointer semantics.

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

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

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

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

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

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

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

Listing A.11: 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_;
    };
    boost::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 boost::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 boost::shared_ptr<Type>&);
};
```


because it would make it difficult to implement observability. Instead, a class template was provided for this purpose which is called `Handle`. Its implementation, shown in listing A.11, 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. The desired behavior is obtained almost for free; 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.

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. 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.[9] The link should only be changed from the original handle—the master 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 master 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 `Handle` instance is returned from an inspector, there's no way to downcast it to `RelinkableHandle`.

A.8. Design patterns

A few design patterns were implemented in QuantLib. I refer the reader to the Gang of Four book [12] for a description of such patterns; in this section, I will limit myself to pointing out in which ways our implementations were tailored to the requirements of the library.

[9] This is not as far-fetched as it might seem; we've been bitten by it.

A.8.1. The Observer pattern

The use of the Observer pattern in the QuantLib library is widespread; to cite just one example, chapter 2 describes at length how financial instruments use it to keep track of changes that should cause them to recalculate their values.

The implementation in the QuantLib library (whose interface is shown in listing A.12) is close enough to that described in the Gang of Four book; but as often happens with patterns, even a straightforward implementation has a number of possible points of customizations. I briefly describe four of them, namely, passing of information, lifetime management, copy behavior, and exception management.

The information passed to observers is very simple—indeed, all they get to know is that something changed. Although it would have been possible to provide more information (namely, by having the `update` method take the notifying observable as an argument) we felt that this feature was not worth the added complexity. A few observables might have used it to save a few cycles by only recalculating some of their data; however, this would have required quite a bit of housekeeping (and in the case of instruments, it would have prevented, or at least disrupted, the common machinery for lazy calculation described in chapter 1.)

The second problem was to make sure that the observer and observables lifetimes were properly synchronized, thus ensuring that no notification is sent (with rather disastrous results) to an already deleted object. This was accomplished by letting observers store shared pointers to their observables. On the one hand, this ensures that no observable is deleted before an observer is done with it. On the other hand, it makes it possible for observers to unregister with any observable before being deleted; in turn, this makes it safe for observables to store a list of raw pointers to their observers, as such raw pointers will be removed from the list before they can become dangling.

Onwards to the third issue. As once noted by G. K. Chesterton [5],

> Poets have been mysteriously silent on the subject of cheese.

The Gang of Four was just as silent on the subject of 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.

Finally, the possibility had to be considered of an observer raising 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. Were the exception to propagate, the loop would be aborted and a number of observers would not receive the notification. Our solution—admittedly

Listing A.12: Interface of the `Observable` and `Observer` classes.

```
class Observable {
    friend class Observer;
  public:
    Observable();
    Observable(const Observable&);
    Observable& operator=(const Observable&);
    virtual ~Observable();
    void notifyObservers();
  private:
    void registerObserver(Observer*);
    void unregisterObserver(Observer*);
    list<Observer*> observers_;
};

class Observer {
  public:
    Observer();
    Observer(const Observer&);
    Observer& operator=(const Observer&);
    virtual ~Observer();
    void registerWith(const shared_ptr<Observable>&);
    void unregisterWith(const shared_ptr<Observable>&);
    virtual void update() = 0;
  private:
    list<shared_ptr<Observable> > observables_;
};
```


an imperfect one—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; however, we felt this to be the least of the two evils.

A.8.2. The Singleton pattern

A.8.3. The Visitor pattern