Implementing QuantLib

Luigi Ballabio

4 Cash flows and coupons

Show me the money!

"CASH IS KING," says a sign in the office of StatPro's CFO.[1] In order to deal with the comings and goings of its Majesty, 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, methods are provided to return the date and amount of the cash flow. To save one from comparing dates explicitly (and for consistency, as explained in the aside on page 60) another convenience method tells whether the cash flow has already occurred at a given date. Finally, a hook is provided so that cash-flow classes can take part in the Acyclic Visitor pattern [17]; an example of its use will be provided in section 4.3.

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[2] and which is reused in other parts of the code. The interfaces of the two classes are shown in listing 4.1.

The library provides a simple implementation of the interface in the aptly named `SimpleCashFlow` class. It's boring enough not to be shown 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.

[1] I trust I'm not revealing any secret business practice.

[2] 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.

Listing 4.1: 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&);
};
```

4.2. Interest-rate coupons

The `Coupon` class (shown in listing 4.2) can be used as a parent class for any cash-flow based on accruing an interest rate over a given period and based on a given day-count convention. It is of course an abstract base class; it defines additional interface for any such cash flow, and implements a few methods dealing with date calculations.

The abstract interface includes a `rate` method, which in derived classes will return (of course) 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`, a case could be made for storing the day counter as a data member of the `Coupon` class; as discussed in section 3.1, 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

Listing 4.2: 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_;
};
```

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 apropriate boolean flag to `hasOccurred`.

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.[3]

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 section A.2 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.[4] 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 documenting the issue.

[3] The same could be said for the `amount` method; it could, too, have a default implementation.

[4] The only derived classes that wouldn't need to store the coupon dates as data members would probably be those decorating an existing coupon. As I write, there's no such class in the library.

Listing 4.3: 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_;
};
```


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 listing 4.3. 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; as you know, premature generalization is evil) to support different compounding rules.

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 listing 4.4) has a number of issues that I'll point out as I describe it.[5]

The first issue might be the name itself: `FloatingRateCoupon` suggests a particular type of coupon, i.e., that paying some kind of LIBOR rate. However, the class is more general that 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[6] as well as other details of the fixing; namely, the

[5] Unless they can be fixed without breaking backward compatibility, we'll have to live with any shortcomings until the yet unplanned release 2.0.

[6] 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.

Listing 4.4: 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;
    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
};
```

Listing 4.4 (continued.)

```
 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());
}
```

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.[7]

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 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.

[7] 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.

The last methods shown deal with a possible convexity adjustment for the index fixing. The `adjustedFixing` method is written in terms of the `rate` method and inverts the $R = g \cdot \hat{F} + s$ 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.

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.[8] 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,)

[8] 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.

Aside: keeping one's balance.

The loss of generality I mentioned about the `FloatingRateCoupon` class is, 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.

Listing 4.5: 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
};
```


Like we did for `Instrument`, we used the Strategy pattern to code our solution. However, the pattern 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 listing 4.5. 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.[9] 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.

The `FloatingRateCoupon` class defines a `setPricer` method that takes a pricer instance and stores it into the coupon.[10] Finally, the `rate` method can be defined as shown in listing 4.4: 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 firthermore, 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

[9] The class declares other methods, not shown here.

[10] The implementation is not as simple as that, but it does not concern us here.

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 listing 4.6. 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,[11] 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 wil be done by pricers. Listing 4.7 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.

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

[11] The *Ibor* name was generalized from the common suffix of EURIBOR, LIBOR, and a number of like indexes.

Listing 4.6: 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 boost::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, inArrears) {}
};
```


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; I'll return to them in a later example.

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 way (shown in listing 4.8 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. 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

Listing 4.7: 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_;
};
```

Listing 4.8: 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();
    }
};
```

Listing 4.9: 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);
    }
};
```

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.

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 is to implement a par-coupon pricer explicitly. As shown in listing 4.9, 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.

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.[12] 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 on page 65 when I described the `adjustedFixing` method.

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 interface of the resulting `CappedFlooredCoupon` class is shown in listing 4.10; its implementation in listing 4.11.

[12] The same problem must be faced when overriding the coupon's `rate` method.

Listing 4.10: Interface 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_;
};
```


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, it may also be noted 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 just 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 due on the one hand, to the so-called *forwarding problem* [9]; 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.

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 we can note. 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 just 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

Listing 4.11: Implementation of the `CappedFlooredCoupon` class.

```
CappedFlooredCoupon::CappedFlooredCoupon(
        const boost::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;
}
```

Listing 4.11 (continued.)

```
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);
}
```

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 maintain 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 listing 4.12 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 argu-

Listing 4.12: 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) {}
};
```


ments 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.

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. More often than not, 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 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 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> notionals(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 we want 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 notionals 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.[13]

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[14] and was recently made popular in the dynamic-language camp by Martin Fowler (of *Refactoring* fame) under the name of Fluent Interface. The idea is simple enough. Instead of a function, we'll use an object to build our coupons. Instead of passing the parameters together in a single call, we'll pass them one at a time to aptly named

[13] 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.

[14] It is described as an established idiom in *C++ FAQs* , published in 1998.

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 listing 4.13. The constructor takes just a couple of arguments, namely, those that are both required and not ambiguous: i.e., 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,[15] 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.

[15] For N parameters we'll need $2N$ overloads, which is much more manageable than the alternative.

Listing 4.13: 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;
  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_;
};
```

Listing 4.13 (continued.)

```
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 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 interesting exercise for the eager reader might be to generate a sequence of several cash flows obtaining their amounts from a single Monte Carlo simulation.[16]

On the other hand, the same design can be used for other kinds of coupons besides interest-rate ones, which was done recently (as I write this) 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 (namely, a third and so-far unknown kind of coupon) before trying a refactoring.

4.3. Cash-flow analysis

After having generated one or more legs, we naturally want to analyse 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 listing 4.14. 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.

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.

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

[16] 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.

Listing 4.14: 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;
}
```

might be the most convenient choice. My advice (which I'm going to disregard it 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.[17]

The implementation of the `bps` method is sketched in listing 4.15. 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.

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,

[17] 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.

Listing 4.15: 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) {}
        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();
}
```

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, a user will expect to retrieve a great many values from a bond instance: clean and dirty prices, accrued amount, yield, whatever suits a trader's fancy. The `Bond` class will need methods to return such values. Part of the resulting interface is shown in listing 4.16; you can look at the class declaration in the library for a full list of the supported methods.

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.[18] The `settlementValue` method, shown in listing 4.17 with the others I'll be discussing here, is like most methods relying on

[18] The NPV is distinct from the settlement value in that the former discounts the cash flows to the reference date of the discount curve.

Listing 4.16: 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_;
};
```

Listing 4.16 (continued.)

```
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;
};
```


a pricing engine: it triggers calculation if needed, checks that the engine assigned a value to the data member, and returns it.

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

Listing 4.17: 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;

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

Listing 4.18: 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_;
};
```


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 listing 4.16. 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 listing 4.18. 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.

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 listing 4.19. It

Listing 4.19: 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);
    }
};
```

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.

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 advantage that all calculations are performed, including slower ones such as the yield calculation (which needs iterative root solving.) This would not be convenient, say, if one just wanted bond prices.[19]

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.

[19] 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.