Implementing QuantLib

Luigi Ballabio

3 Term structures

The shape of things to come

CHANGE IS the only constant, as Heraclitus said. Paradoxically, the aphorism still holds after twenty-five centuries; also in quantitative finance, where most—if not all—quantities obviously vary over time.

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

3.1. The `TermStructure` class

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

3.1.1. Interface and requirements

Once abstracted out, the base term-structure class (whose interface is shown in listing 3.1) was responsible for three basic tasks. The first is to keep track of its own reference date, i.e., the date at which—in a manner of speaking—the future begins.[1] For a volatility term structure, that would most likely be today's date. For a yield curve, it might be today's date; but depending on the conventions used at one's desk (for instance, an interest-rate swap desk whose deals are all settled on the second business day) the reference date might be the result of advancing today's date by a few settlement days. Our term-structure class must be able to perform such a calculation if needed. Also, there might be cases in which the reference date is specified externally (such as when a sequence of dates, including the reference, is tabulated somewhere together with the corresponding discount factors.) Finally, the calculation of the reference date might be altogether delegated to some other object; we'll see such an arrangement in section 3.2.4. In

[1]This is not strictly true of all term structures. However, we'll leave it at that for the time being.

Listing 3.1: Interface of the `TermStructure` class.

```
class TermStructure : public virtual Observer,
                      public virtual Observable,
                      public Extrapolator {
  public:
    TermStructure(const DayCounter& dc = DayCounter());
    TermStructure(const Date& referenceDate,
                  const Calendar& calendar = Calendar(),
                  const DayCounter& dc = DayCounter());
    TermStructure(Natural settlementDays,
                  const Calendar&,
                  const DayCounter& dc = DayCounter());
    virtual ~TermStructure();

    virtual DayCounter dayCounter() const;
    virtual Date maxDate() const = 0;
    virtual Time maxTime() const;
    virtual const Date& referenceDate() const;
    virtual Calendar calendar() const;
    virtual Natural settlementDays() const;

    void update();
  protected:
    Time timeFromReference(const Date& date) const;

    void checkRange(const Date&, bool extrapolate) const;
    void checkRange(Time, bool extrapolate) const;
    bool moving_;
};
```

all these cases, the reference date will be provided to client code by means of the `referenceDate` method. The related `calendar` and `settlementDays` methods return the calendar and number of settlement days, if any, used for the calculation.

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

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

3.1.2. Implementation

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

Three constructors are available. One simply takes a day counter (used for time calculations, as we will see later) but no arguments related to reference-date calculation. The resulting term structure has no means to calculate such date; therefore, derived classes calling this constructor must take care of the calculation by overriding the virtual `referenceDate` method. The implementation sets `moving_` to `false` and `updated_` to `true` in order to inhibit further calculations in the base class.

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

Finally, a third constructor takes a number of settlement days and a calendar. When this one is used, the reference date will be calculated as today's date advanced by the given number of business days according to the given calendar. Besides copying the passed data to the corresponding data members, the implementation sets `moving_` to `true` and `updated_` to `false` (since no calculation is performed at this time.) However, that's not the full story; if today's date changes, the term structure must be notified so that it can update its reference date. The `Settings` class (described in section A.6) provides global access to the current evaluation date, with which the term structure registers as an observer. When a change is notified, the `update` method is executed. If the reference date is moving, the body of the method sets `updated_` to `false` before forwarding the notification to the term structure's own observers.

Apart from trivial inspectors such as the `calendar` method, the implementation of the first task is completed with the `referenceDate` method. If the reference date needs to be calculated, it does so by retrieving the current evaluation date,

[2]Before reading this section, you might want to skim briefly through section A.2 for a description of the classes used for date- and time-related calculations.

Listing 3.2: Implementation of the `TermStructure` class.

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

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

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

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

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

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

Listing 3.2 (continued.)

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

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

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

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

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

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

advancing it as specified, and storing the result in the referenceDate_ data member before returning it. This moves the term structure to the newly-specified evaluation date.

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

You might object that this is, to use the term coined by Kent Beck [11], a code smell. A term-structure instance might store a day counter or a reference date (or likely both) not corresponding to the actual ones used by its methods. This disturbed me as well; and indeed, earlier versions of the class declared the dayCounter method as purely virtual and did not include the data member. However, it is a necessary evil in the case of the reference date, since we need a data member to cache its calculated value. Due to the broken-window effect [13], the day counter, calendar and settlement days followed (after a period in which we developed a number of derived term structures, all of which had to define the same data members.)

A final question remains: what day counter should be used? Fortunately, it doesn't matter much. If one is only working with dates (i.e., provides dates as an input for the construction of the term structure and uses dates as arguments to retrieve values) the effects of choosing a specific day counter will cancel out as long as the day counter is sufficiently well behaved: for instance, if it is homogeneous (by which I mean that the time $T(d_1, d_2)$ between two dates d_1 and d_2 equals the time $T(d_3, d_4)$ between d_3 and d_4 if the two pairs of dates differ by the same number of days) and additive (by which I mean that $T(d_1, d_2) + T(d_2, d_3)$ equals $T(d_1, d_3)$ for all choices of the three dates.) Two such day counters are the actual/360 and the actual/365-fixed ones. Similarly, if one is only working with times, the day counter will not be used at all.

Onwards with the third and final task. The job of defining the valid date range is delegated to derived classes, which must define the maxDate method (here declared as purely virtual.) The corresponding time range is calculated by the maxTime method, which simply converts the latest valid date to time by means of the timeFromReference method. Finally, the two checkRange methods implement the actual range checking (the one taking a date by forwarding the request to the other after converting it to a time) and throw an exception if the passed argument is not in the valid range. The check can be overridden by a request

Aside: evaluation date tricks.

If no evaluation date is set, the `Settings` class defaults to returning today's date. Unfortunately, the latter will change silently (that is, without notifying its observers) at the strike of midnight, causing mysterious errors. If you run overnight calculations, you'll have to perform the same feat as Hiro Nakamura in *Heroes*—freeze time. Explicitly settings today's date as the evaluation date will keep it fixed, even when today becomes tomorrow.
Another trick worth knowing: if all your term structures are moving, setting the evaluation date to tomorrow's date (while keeping everything else unchanged, of course) and recalculating the value of your instruments will give you the daily theta of your portfolio.

to extrapolate outside the domain of the term-structure; this can be done either by passing an optional boolean argument to `checkRange` or by using the facilities provided by the `Extrapolator` class (see section A.5) from which `TermStructure` inherits. Extrapolation is only allowed beyond the maximum date; requests for dates before the reference date are always rejected.

3.2. Yield term structures

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

3.2.1. Interface and implementation

The interface of the `YieldTermStructure` class is sketched in listing 3.3. The constructors just forward their arguments to the corresponding constructors in the `TermStructure` class—nothing to write home about. The other methods return information on the yield structure in different ways; on the one hand, they can return zero rates, forward rates, and discount factors;[3] on the other hand, they are overloaded so that they can return information as function of either dates or times.

Of course, there is a relationship between zero rates, forward rates, and discount factors; the knowledge of any one of them is sufficient to deduce the others (I won't bore you with the formulas here—you know them.) This is reflected in the implementation, outlined in listing 3.4; the Template Method patterns is used to implement all public methods directly or indirectly in terms of the protected

[3]Rates are returned as instances of the `InterestRate` class, described briefly in section A.4.

Listing 3.3: Partial interface of the YieldTermStructure class.

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

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

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

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

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

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

Listing 3.4: Partial implementation of the `YieldTermStructure` class.

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

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

`discountImpl` abstract method. Derived classes only need to implement the latter in order to return any of the above quantities.

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

What if the author of a derived class doesn't want to implement `discountImpl`, though? After all, one might want to describe a yield curve in terms, say, of zero rates. Ever ready to serve (just like Jeeves in the P. G. Wodehouse novels—not that you're Bernie Wooster, of course) QuantLib provides a couple of classes to be used in this case. The two classes (outlined in listing 3.5) are called `ZeroYieldStructure` and `ForwardRateStructure`. They use the Adapter pattern[4] to transform the discount-based interface of `YieldTermStructure` into interfaces based on zero-yield and instantaneous-forward rates, respectively.

The implementation of `ZeroYieldStructure` is simple enough. A few constructors (not shown here) forward their arguments to the corresponding constructors in the parent `YieldTermStructure` class. The Adapter pattern is implemented in the protected section: an abstract `zeroYieldImpl` method is declared and used to implement the `discountImpl` method. Thus, authors of derived classes only need to provide an implementation of `zeroYieldImpl` to obtain a fully functional

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

Listing 3.5: Outline of the ZeroYieldStructure and ForwardRateStructure classes.

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

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

yield curve.[5] Note that, due to the formula used to obtain the discount factor, such method must return zero-yields as continuously-compounded annualized rates.

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

[5]Of course, the other required methods (such as maxDate) must be implemented as well.

Aside: symmetry break.

You might argue that, as in George Orwell’s *Animal Farm*, some term structures are more equal than others. The discount-based implementation seems to have a privileged role, being used in the base `YieldTermStructure` class. A more symmetric implementation might define three abstract methods in the base class (`discountImpl`, `zeroYieldImpl`, and `forwardImpl`, to be called from the corresponding public methods) and provide three adapters, adding a `DiscountStructure` class to the existing ones.
Well, the argument is sound; in fact, the very first implementation of the `YieldTermStructure` class was symmetric. The switch to the discount-based interface and the reasons thereof are now lost in the mists of time, but might have to do with the use of `InterestRate` instances; since they can require changes of frequency or compounding, `zeroYield` (to name one method) wouldn’t be allowed to return the result of `zeroYieldImpl` directly anyway.

the zero yields, one can inherit from `ZeroYieldStructure` and be done with it; however, it is conceptually cleaner to express the curve in terms of the forwards if they were the actual focus of the model.

The two adapter classes I just described and the base `YieldTermStructure` class itself were used to implement interpolated discount, zero-yield, and forward curves. Listing 3.6 outlines the `InterpolatedZeroCurve` class template; the other two (`InterpolatedForwardCurve` and `InterpolatedDiscountCurve`) are implemented in the same way.

The template argument `Interpolator` has a twofold task. On the one hand, it acts as a traits class [18]. It specifies the kind of interpolation to be used as well as a few of its properties, namely, how many points are required (e.g., at least two for a linear interpolation) and whether the chosen interpolation is global (i.e., whether or not moving a data point changes the interpolation in intervals that do not contain such point; this is the case, e.g., for cubic splines.) On the other hand, it doubles as a poor man’s factory; when given a set of data points, it is able to build and return the corresponding `Interpolation` instance.[6]

The public constructor takes the data needed to build the curve: the set of dates over which to interpolate, the corresponding zero yields, the day counter to be used, and an optional `interpolator` instance. For most interpolations, the last parameter is not needed; it can be passed when the interpolation needs parameters. The implementation forwards to the parent `ZeroYieldStructure` class the first of the passed dates, assumed to be the reference date for the curve, and the day

[6]The `Interpolation` class is described in section A.5 together with a few available interpolations and the corresponding traits.

Listing 3.6: Outline of the InterpolatedZeroCurve class template.

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

        // convert dates_ into times_

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

Aside: twin classes.

If you looked at the code for the interpolated discount and forward curves, you surely noted how similar they are to the interpolated zero-yield curve described here. The question naturally arises: would it be possible to abstract out common code? Or maybe we could even do with a single class template?
The answers are yes and no, respectively. Some code could be abstracted in a template class (in fact, this has been done since this chapter was written.) However, the curves must implement three different abstract methods (`discountImpl`, `forwardImpl`, and `zeroYieldImpl`) so we still need all three classes as well as the one containing the common code.

counter; the other arguments are stored in the corresponding data members. After performing a few consistency checks, it converts the dates into times (using, of course, the passed reference date and day counter,) asks the interpolator to create an `Interpolation` instance, and stores the result.

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

Finally, let me note that the `InterpolatedZeroCurve` class defines a few protected constructors. They take the same arguments as the constructors of the parent `ZeroYieldStructure` class, as well as an optional `Interpolator` instance. The constructors forward the needed arguments to the corresponding parent-class constructors, but do not create the interpolation—they cannot, since they don't take any zero-yield data. They are defined so that it is possible to inherit from `InterpolatedZeroCurve`; derived classes will provide the data and create the interpolation based on whatever arguments they take (an example of this will be shown in the remainder of this section.) For the same reason, most data members are declared as mutable; as noted in the aside on page 10, this makes it possible for derived classes to update the interpolation lazily, should their data change.

3.2.3. Example: bootstrapping an interpolated yield curve

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

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

The implementation of our class template is shown in listing 3.7. The class takes three template arguments. The first two determine the choice of the underlying data and the interpolation method, respectively; our goal is to instantiate the template as, say,

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

or some such combination. I'll refer to the first parameter as the *bootstrap traits*; the second is the interpolator already described on page 41. The third, and seemingly ungainly, parameter specifies a class which implements the bootstrapping algorithm. The parameter has a default value (the IterativeBootstrap class, which I'll describe later) so you can happily forget about it most of the times; it is provided so that interested developers can replace the bootstrapping algorithm with another one, either provided by the library or of their own creation. For those not familiar with its syntax, I'll mention that it's a template template parameter [22]. The repetition is not an error, as suspected by my spell checker as I write these lines; it means that the parameter should be an uninstantiated class template (in this case, one taking a single template argument) rather than a typename.[7]

Before declaring the class interface, we need another bit of template programming to determine the parent class of the curve. On the one hand, we inherit our term structure from the LazyObject class; as described in chapter 2, this will enable the curve to re-bootstrap itself when needed. On the other hand, we want to inherit it from one of the interpolated curves described in section 3.2.2; depending on the choice of underlying data (discount, zero yields, or forward rates,) we must select one of the available class templates and instantiate it with the chosen interpolator class. This is done by storing the class template in the bootstrap traits. Unluckily, C++ doesn't allow template typedefs at this time.[8] Therefore, the traits define an inner class template curve which takes the interpolator as its template parameter and defines the instantiated parent class as a typedef; this can be seen in the definition of the Discount traits, partially shown in listing 3.8.

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

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

that we can add to the list of parent classes.[9] For the instantiation shown as

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

[8]Template aliases will probably be introduced in the upcoming revision of the C++ standard.

[9]For those unfamiliar with the dark corners of template syntax, the template keyword in the expression is a hint for the compiler. When reading this expression, the compiler doesn't know what

Listing 3.7: Implementation of the `PiecewiseYieldCurve` class template.

```
template <class Traits, class Interpolator,
          template <class> class Bootstrap = IterativeBootstrap>
class PiecewiseYieldCurve
    : public Traits::template curve<Interpolator>::type,
      public LazyObject {
  private:
    typedef typename Traits::template curve<Interpolator>::type
                                                       base_curve;
    typedef PiecewiseYieldCurve<Traits,Interpolator,Bootstrap>
                                                       this_curve;
    typedef typename Traits::helper helper;
  public:
    typedef Traits traits_type;
    typedef Interpolator interpolator_type;
    PiecewiseYieldCurve(
           Natural settlementDays,
           const Calendar& calendar,
           const std::vector<shared_ptr<helper> >& instruments,
           const DayCounter& dayCounter,
           Real accuracy = 1.0e-12,
           const Interpolator& i = Interpolator());

    // inspectors not shown

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

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

Listing 3.7 (continued.)

```
template <class T, class I,
          template <class> class B>
PiecewiseYieldCurve<T,I,B>::PiecewiseYieldCurve(
           Natural settlementDays,
           const Calendar& calendar,
           const std::vector<shared_ptr<helper> >& instruments,
           const DayCounter& dayCounter,
           Real accuracy,
           const I& interpolator)
: base_curve(settlementDays, calendar, dayCounter, i),
  instruments_(instruments), accuracy_(accuracy) {
    bootstrap_.setup(this);
}

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

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

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

Listing 3.8: Sketch of the `Discount` bootstrap traits.

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

    // other static methods
}
```

an example on page 44, with `Discount` as bootstrap traits and `LogLinear` as interpolator, the above works out as

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

which in turn corresponds—as desired—to

```
InterpolatedDiscountCurve<LogLinear>
```

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

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

The actual interface, at last. The constructors (of which only one is shown in the listing) take the arguments required for instantiating the parent interpolated curve, as well as a vector of helpers and the target accuracy for the bootstrap. Next, a number of inspectors (such as `times` or `data`) are defined, overriding the versions in the parent class; the reason for this will be explained later on. The public interface is completed by the `update` method.

The protected methods include `performCalculations`, needed to implement the `LazyObject` interface; and `discountImpl`, which (like the public inspectors)

`Traits` is and has no means to determine that `Traits::curve` is a class template. Adding the keyword gives it the information, required for processing the rest of the expression correctly.

Aside: a friend in need.

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

overrides the parent-class version. The `Bootstrap` class template is instantiated with the type of the curve being defined. Since it will need access to the internals of the curve, the resulting class is declared as a friend of the `PiecewiseYieldCurve` class; the same is done for the `BootstrapError` class, used in the bootstrap algorithm and described later. Finally, we store an instance of the bootstrap class, as well as the required curve accuracy and the helpers.

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

The `performCalculations` simply delegates its work to the `Bootstrap` instance. Lastly, a look at the `discountImpl` method shows us why such method had to be overridden; before calling the parent-class implementation, it has to ensure that the data were bootstrapped by calling the `calculate` method. This also holds for the other overridden inspectors, all following the same pattern.

At this point, I need to describe the `BootstrapHelper` class template; its interface is sketched in listing 3.9. For our curve, we'll instantiate it (as you can see in the `Discount` traits shown earlier) as `BootstrapHelper<YieldTermStructure>`;

Listing 3.9: Interface of the BootstrapHelper class template.

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

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

    virtual void setTermStructure(TS*);

    virtual Date latestDate() const;

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

for convenience, the library provides an alias to this class called `RateHelper` that can be used in place of the more verbose type name.

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

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

Listing 3.10: Sketch of the `IterativeBootstrap` class template.

```
template <class Curve>
class IterativeBootstrap {
    typedef typename Curve::traits_type Traits;
    typedef typename Curve::interpolator_type Interpolator;
  public:
    IterativeBootstrap();
    void setup(Curve* ts);
    void calculate() const;
  private:
    Curve* ts_;
};

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

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

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

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

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

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

Listing 3.10 (continued.)

```
    Brent solver;

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

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

          // bracket the solution
          Real min = Traits::minValueAfter(i, ts_->data_);
          Real max = Traits::maxValueAfter(i, ts_->data_);

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

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

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

Two more methods are used for setting up the bootstrap algorithm. The `setTermStructure` method links the helper with the curve being built. The `latestDate` method returns the latest date for which curve data are required in order to calculate the implied value of the market datum;[10] such date will be used as the coordinate of the node being bootstrapped. The last method (`update`) forwards notifications from the quote to the observers of the helper.

The library provides a few concrete helper classes inherited from `RateHelper`. In the interest of brevity, allow me to do a little hand-waving here instead of showing you the actual code. Each of the helper classes implements the `impliedValue` for a specific instrument and includes code for returning the proper latest date. For instance, the `DepositRateHelper` class forecasts a quoted deposit rate by asking the curve for the forward rate between its start and maturity dates; whereas the `SwapRateHelper` class forecasts a swap rate by instantiating a `Swap` object, pricing it on the curve being bootstrapped, and implying its fair rate.

We can finally dive into the bootstrap code. Listing 3.10 shows the interface of the `IterativeBootstrap` class, which is provided by the library and used by default. For convenience, typedefs are defined to extract from the curve the traits and interpolator types. The constructor and the `setup` method are not of particular interest. The first just initializes the contained term-structure pointer to a null one; the second stores the passed curve pointer, checks that we have enough helpers to bootstrap the curve, and registers the curve as an observer of each helper. The bootstrap algorithm is implemented by the `calculate` method. In the version shown here, I’ll gloss over a few details and corner cases; if you’re interested, you can peruse the full code in the library.

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

At this point, we can instantiate the one-dimensional solver that we’ll use at each node (more details on this in appendix A) and start the actual bootstrap. The calculation is written as two nested loops; an inner one—the bootstrap proper—that walks over each node, and an outer one that repeats the process. Iterative

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

[11]Of course, the same helpers could not be passed safely to different curves if a multi-threaded environment. Then again, much of QuantLib is not thread-safe.

Listing 3.11: Interface of the `BootstrapError` class template.

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

bootstrap is needed when a non-local interpolation (such as cubic splines) is used. In this case, setting the value of a node modifies the whole curve, invalidating previous nodes; therefore, we must go over the nodes a number of times until convergence is reached.

As I mentioned, the inner loop walks over each node—starting, of course, from the one at the earliest date. During the first iteration (when `iteration == 0`) the interpolation is extended a point at a time; later iterations use the full data range, so that the previous results are used as a starting point and refined. After each node is added, a one-dimensional root-finding algorithm is used to reproduce the corresponding market quote. For the first iteration, a guess for the solution can be provided by the bootstrap traits or by extrapolating the curve built so far; for further iteration, the previous result is used as the guess. The maximum and minimum value are provided by the traits, possibly based on the nodes already bootstrapped. For instance, the traits for bootstrapping zero or forward rates can prevent negative values by setting the minimum to a zero; the traits for discount factors can do the same by ensuring that the discounts are not increasing, i.e., by setting the maximum to the discount at the previous node.

The only missing ingredient for the root-finding algorithm is the function whose zero must be found. It is provided by the `BootstrapError` class template (shown in listing 3.11,) that adapts the helper's `quoteError` calculation to a function-object interface. Its constructor takes the curve being built, the helper for the current node, and the node index, and stores them. Its `operator()` makes instances of this class usable as functions; it takes a guess for the node value, modifies the curve data accordingly, and returns the quote error.

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

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

This concludes the bootstrap; and, as I don't want to further test your patience, it also concludes the example. Sample code using the `PiecewiseYieldCurve` class can be found in the QuantLib distribution, namely, in the swap-valuation example.

3.2.4. Example: adding z-spread to a yield curve

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

The implementation of the `ZeroSpreadedTermStructure` is shown in listing 3.12. As previously mentioned, it is based on zero-yield rates; therefore, it inherits from the `ZeroYieldStructure` adapter described in section 3.2.2 and will have to implement the required `zeroYieldImpl` method. Not surprisingly, its constructor takes as arguments the risk-free curve to be modified and the z-spread to be applied; to allow switching data sources, both are passed as handles. The arguments are stored in the corresponding data members, and the curve registers with both of them as an observer. The `update` method inherited from the base class will take care of forwarding any received notifications.

It should be noted that none of the base-class constructors was called explicitly. As you might remember from section 3.1.2, this means that our curve stores no data that can be used by the `TermStructure` machinery; therefore, it must provide its own implementation of the methods related to reference-date calculation. In true Decorator fashion, this is done by delegating behavior to the wrapped object; each of the `referenceDate`, `dayCounter`, `calendar`, `settlementDays`, and `maxDate` methods forwards to the corresponding method in the risk-free curve.

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

Listing 3.12: Implementation of the ZeroSpreadedTermStructure class.

```
class ZeroSpreadedTermStructure : public ZeroYieldStructure {
  public:
    ZeroSpreadedTermStructure(
                  const Handle<YieldTermStructure>& h,
                  const Handle<Quote>& spread);
    : originalCurve_(h), spread_(spread) {
        registerWith(originalCurve_);
        registerWith(spread_);
    }
    const Date& referenceDate() const {
        return originalCurve_->referenceDate();
    }
    DayCounter dayCounter() const {
        return originalCurve_->dayCounter();
    }
    Calendar calendar() const {
        return originalCurve_->calendar();
    }
    Natural settlementDays() const {
        return originalCurve_->settlementDays();
    }
    Date maxDate() const {
        return originalCurve_->maxDate();
    }
  protected:
    Rate zeroYieldImpl(Time t) const {
        InterestRate zeroRate =
            originalCurve_->zeroRate(t, Continuous,
                                     NoFrequency, true);
        return zeroRate + spread_->value();
    }
  private:
    Handle<YieldTermStructure> originalCurve_;
    Handle<Quote> spread_;
};
```

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

3.3. Other term structures

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

3.3.1. Default-probability term structures

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

Unlike yield term structures (in which all methods are implemented in terms of the `discountImpl` method) the base default-probability structure has no single method for others to build upon. Instead, as shown in listing 3.13, it declares two abstract methods `survivalProbabilityImpl` and `defaultDensityImpl`. It’s left to derived classes to decide which one should be implemented in terms of the other; the base class returns `survivalProbability` and `defaultDensity` based on the respective implementation methods,[12] `defaultProbability` based trivially on `survivalProbability`, and `hazardRate` in terms of both survival probability and default density.

Listing 3.14 sketches the adapter classes that, as for yield term structures, allow one to define a new default-probability structure in terms of the single quantity of his choice—either survival probability, default density, or hazard rate (the default probability is so closely related to survival probability that we didn’t think it necessary to provide a corresponding adapter.) The first one, `SurvivalProbabilityStructure`, defines `defaultDensityImpl` in terms of the implementation of `survivalProbabilityImpl`, which is left purely virtual and must be provided in derived classes; the second one, `DefaultDensityStructure`, does the opposite; and the last one, `HazardRateStructure`, defines both survival probability and default density in terms of a newly-declared purely abstract `hazardRateImpl` method.

Unfortunately, some of the adapters rely on numerical integration in order to provide conversions among the desired quantities. The provided implementations use dark magic in both maths and coding (namely, Gaussian quadratures and

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

Listing 3.13: Sketch of the `DefaultProbabilityTermStructure` class.

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

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

    Probability defaultProbability(Time t) const {
        return 1.0 - survivalProbability(t);
    }
    Probability defaultProbability(Time t1,
                                   Time t2) const {
        Probability p1 = defaultProbability(t1),
                    p2 = defaultProbability(t2);
        return p2 - p1;
    }

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

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

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

Listing 3.14: Adapter classes for default-probability term structures.

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

class DefaultDensityStructure
    : public DefaultProbabilityTermStructure {
  public:
    ...constructors...
  protected:
    Probability survivalProbabilityImpl(Time t) const {
        // returns 1 minus the integral of the default density from 0 to t
    }
};

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

Aside: Cinderella method.

In the implementation of `DefaultProbabilityTermStructure`, you've probably noticed another symmetry break like the one discussed in the aside on page 41. There's a difference though; in that case, discount factors were singled out to be given a privileged role. In this case, hazard rates are singled out to play the mistreated stepsister; there's no `hazardRateImpl` beside the similar methods declared for survival probability or default density. Again, a look at past versions of the implementation shows that once, it was symmetric; and again, I can give no reason for the change—although I'm sure it looked like a good idea at the time.
The effect is that classes deriving from the `HazardRateStructure` adapter must go through some hoops to return hazard rates, since they're not able to call `hazardRateImpl` directly; instead, they have to use the default implementation and return the ratio of default density and survival probability (possibly performing an integration along the way.) Unfortunately, even our fairy godmother can't change this now without risking to break existing code.

`boost::bind`) to perform the integrations efficiently; but when inheriting from such classes, you should consider overriding the adapter methods if a closed formula is available for the integral.

As for yield curves, the library provides a few template classes that implement the adapter interfaces by interpolating discrete data, as well as a generic piecewise default-probability curve template and the traits required to select its underlying quantity. Together with the existing interpolation traits, this allows one to instantiate classes such as `PiecewiseDefaultCurve<DefaultDensity,Linear>`. The implementation is quite similar to the one described in section 3.2.3; in fact, so much similar that it's not worth describing here. The only thing worth noting is that the default-probability structure is not self-sufficient: in order to price the instruments required for its bootstrap, a discount curve is needed. You'll have to be consistent and use the same curve for your pricing engines; otherwise, you might suddenly find out that your CDS are no longer at the money.

3.3.2. Inflation term structures

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

The most noticeable difference is that we have two separate kinds of inflation term structures (and two different interfaces) instead of a single one. The library does provide a single base class `InflationTermStructure`, that contains some inspectors and some common behavior; however, the interfaces returning the actual inflation rates are declared in two separate child classes, leading to the hierarchy sketched in listing 3.15. The two subclasses model zero-coupon and year-on-year inflation rates, which are not easily converted into one another and thus foil our usual multiple-adapter scheme.

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

Other differences are due to the specific quirks of inflation fixings. Since inflation figures for a given month are announced after an observation lag, inflation term structures have a base date, as well as a reference date; the base date is the one corresponding to the latest announced fixing. If an inflation figure is needed for a date in the past with respect to today's date but after the base date, it must be forecast.[14] Also, since inflation fixings are affected by seasonality, inflation term structures provide the means to store an instance of the polymorphic `Seasonality` class (which for brevity I won't describe.) If given, such instance models the desired seasonal correction for the returned inflation rates, in a plain implementation of the Strategy pattern.

Unlike other kinds of term structure, the inflation interface doesn't provide methods taking a time value instead of a date; the whole fixing machinery depends on a number of date calculations (what month we're in, the corresponding period for the fixing, and whatnot) and there's simply no reliable way to convert from times to dates, so we called the whole thing off.

Yet another difference is that inflation term structures store a discount curve. Yes, I made the same face as you when I realized it—and shortly before the 1.0 release, too, when the interfaces would have to be frozen. No, let me rephrase that: I made a worse face than you. In the end, though, I left it alone. It's not the design we used for default-probability structures; however, it has the advantage that a bootstrapped inflation term structure carries around the discount curve it used, so one doesn't have to keep track of their association to be consistent. We can choose the best design for release 2.0, when we have used both for a few years; so all's well that ends well, I guess. Still, I made a mental note to perform code reviews more often.

The remaining machinery (interpolated curves, piecewise bootstrap etc.) is similar to what I described for other kinds of curves. Due to the separate sub-hierarchies for zero-coupon and year-on-year rates, though, there's a couple of differences. On the one hand, there are no adapter classes that implement the whole

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

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

Listing 3.15: Sketch of the `InflationTermStructure` class and its children.

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

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

    Handle<YieldTermStructure> nominalTermStructure() const;

    void setSeasonality(const shared_ptr<Seasonality>&);
  protected:
    Handle<YieldTermStructure> nominalTermStructure_;
    Period observationLag_;
    ...other data members...
};

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

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

interface based on a single underlying quantity—such as, say, `ZeroStructure` for discounts or `HazardRateStructure` for default probabilities. On the other hand, piecewise curves don't use traits to select the underlying quantity (as, for instance, in `PiecewiseYieldCurve<Discount,LogLinear>`.) To select one or the other, you'll have to choose the appropriate class and write something like `PiecewiseZeroInflation<Linear>` instead.

3.3.3. Volatility term structures

3.3.4. Equity volatility structures

3.3.5. Cap and caplet volatility structures

3.3.6. Swaption volatility structures