IMPLEMENTING QUANTLIB の和訳

An Inside Look at Quantitative Finance in C++

Luigi Ballabio

translated by Aki Sakashita

Implementing QuantLib の和訳

Luigi Ballabio と Aki Sakashita

本書はこちらで販売中です http://leanpub.com/implementingquantlib-jp

この版は 2020-02-09 に発行されました。

本書は Leanpub の電子書籍です。Leanpub はリーンパブリッシングプロセスで著者や出版社を支援します。リーンパブリッシング は新しい出版スタイルです。軽量なツールを使って執筆中の電子書籍を出版し、読者のフィードバックをもらいながら魅力的な本に仕上がるまでピボットを繰り返すことができます。

Twitter でシェアしませんか？

本書に関するコメントをTwitter でシェアして Luigi Ballabio と Aki Sakashita を応援してください！

本書のハッシュタグは #quantlib です。

本書に関するコメントを検索する場合は、次のリンクをクリックして下さい。Twitter のハッシュタグを使って検索できます。

#quantlib

Luigi Ballabio 共著

QuantLib Python Cookbook

Implementing QuantLib

図建 QuantLib

Contents

1. はじめに

QuantLib は、様々な金融商品の価格計算アルゴリズムを、オブジェクトモデル化して汎用性と拡張性を持たせたライブラリーです。膨大なライブラリーであり、QuantLib のサイトで得られる Reference Manual や、様々な解説資料を見ても、なかなか全体像はつかめません。またどうやって使うのかもよく判らないと思います。具体的な使い方については、QuantLib をダウンロードした後、Example ディレクトリーいあるいくつかのソースコードをテストする事をお勧めめします。そこでは、具体的な Financial Instrument と Pricing Engine を部品から組み立てて、それらを結合させて、価格計算やリスク量計算のメソッドを呼び出して計算結果を導出する様子が、様々な金融商品を使って紹介されています。

一方で、QuantLib 全体の設計思想とか、主要なモジュールの設計思想などは、ここで紹介する“Implementing QuantLib”を読まれるのがいいかと思います。その Chapter II では、主要モジュールである、Financial Instruments と Pricing Engines の設計思想について解説されています。また Chapter III から Chapter VIII までは、それらの主要部品・モジュールの解説になっています。Chapter III の Term Structure クラスは、Pricing Engine の必須の部品であるイールドカーブを取扱います。また Term Structure クラスは金利だけでなく、Volatility の期間構造も取扱います。Chapter IV の Cash Flows and Coupons は Financial Instrument の主要部品になります。Chapter V から VIII までは、Pricing Engine の主要部品、すなわち価格計算アルゴリズムをカプセル化したオブジェクトモデルです。

さらに、Odds and Ends（付録）で、主要部品ではないものの、日数計算方法、Solver-Optimizer といった計算アルゴリズム、などの周辺部品に関する解説、さらに、QuantLib の中で使われている主要なデザインパターンについての解説が含まれています。

かなり専門的な内容で、理解するには、C++ と金融工学に関する、かなり高いレベルの知識が必要です。（著者本人がイントロダクションの中でそう言ってます。）

日本人にとって、それらを英文で読んで内容を理解するのは、ハードルが高いかも知れません。また同氏は、暗喩として、日本人には馴染みの薄い小説や映画や雑誌のフレーズを随所に使っているので、それもハードルを上げているかも知れません。そこで、著者である Ballabio 氏の了解を得て、日本語の翻訳を行いました。出来るだけ解りやすくする為、直訳せずに、日本語で意味がわかりやすいように、文章の順番を大きく変えたり、意訳したりした部分が多くあります。また専門用語で解りにくいと思われる部分には、“訳注”として、私自身の解説を若干加えています。それでも、完璧な翻訳などありえないので、読んで解りにくい部分は、原典と読み比べて頂きたいと思います。あくまで原典がすべてであり、訳は原典の理解を助ける為のものとしてお読みください。拙訳の為、誤解が生じたりした場合はご容赦下さい。

訳者より

2. Introduction

若さゆえの情熱から、かつて QuantLib の Web サイトにおいて、“標準的で、かつ無料で開放された金融モデルのライブラリー”を目指すと宣言していました。そのような宣言を若干甘めに解釈すれば、ある程度は成功したと言えるかも知れません。そう言う為に、“最初で、(暫くの間かもしれませんが)唯一のオープンソースの金融モデルライブラリー”の定義を都合よく解釈したかも知れませんが。(注:何人かの QuantLib ライブラリーのユーザーの方から、我々のモデルライブラリーを”the QuantLib”と呼んでいただきました。大変光栄な呼称ですが、私としては謙遜して、今の所そう呼ぶのは控えています。)

標準的とかどうかに関わらず、少なくともこれを書いている時点では、新しいバージョンのリリースがあると、毎回、数千件のダウンロードが行われているようです。ユーザーからの寄与も定常的に行われていますし、実務の世界でも使われているようです(金融の世界における秘密主義にもかかわらず、漏れてくる話からの推察ですが)。このプロジェクトの管理者として、全体を総合的に見ると、私は幸せ者であると言っていいかも知れません。

しかし、それが故に、適切な解説書が無いという問題が、より明白になりました。詳細なクラスの解説は提供されていますが(この部分は、自動的に作成する事ができるので、実は簡単でした)、それを読んでも、木を見て森を見ず、の状況に陥るだけで、新しくこれを使ってモデル開発を行おうとするユーザーにとっては、まさにルイス・キャロルの「スナーク狩り」の中で Bellman が述べた言葉に同意をするかもしれません。

> “メルカトル図法の地図で、北極点と赤道、回帰線、子午線は何の役に立つのだ?”と Bellman は叫んだ。それに対し、隊員たちは答えた。“単なる地図記号ですよ”

この本の目的は、その足りない所を埋める事です。この本は、全体のデザインや QuantLib ライブラリーの実装方法のレポートであり、また Mel Brooks の“Young Frankenstein”で極めて特徴的な、“どうやって作ったのか”的な本にも、心情的にはよく似ています(その映画の驚くような成功には及びませんが)。もし読者の方が、既に QuantLib ライブラリーのユーザーであるか、あるいはこれからユーザーになろうとしているのであれば、プログラムコードを読むだけでは解らないような、ライブラリーのデザインについての有用な情報が、この本から得られるでしょう。もし読者がユーザーではないものの、金融工学の世界で働いているのであれば、金融モデルライブラリーのデザインに関する現場報告書として読んで頂いてもかまいません。おそらく、あなたが今直面しているような問題に対して、その解決策と、その理屈も含めて、この本がカバーしている部分がある事に気づくでしょう。読者の方の制約条件によっては、別の解決策を選択するかもしれませんし、あるいはその可能性の方が高いかもしれませんが、この本の中での論点整理から、そういった場合でも得るものがあるかも知れません。

説明の中で、現時点の実装方法の欠点も指摘しようと思います。それは、このライブラリーを蔑む為では無く(私自身もライブラリーの開発に深く関わっていました)、より有用な目的の為です。一つには、既存の弱点を説明する事によって、これから(このライブラリーを使って)モデル開

発しようとする人が同じ失敗をする事を避けられます。もう一方で、このライブラリーをこれからどう向上させていくか道筋を見せてくれるかも知れません。実際にも、すでにそれは起こっており、この本を書く為にプログラムコードを見直していく中で、いくつか弱点の改善を行ってきました。

本のスペースと私の時間の制約から、QuantLib ライブラリーのすべての機能をカバーすることはできません。この本の前半部分は、金融商品や期間構造のオブジェクトモデルといった2〜3種類の最も重要なクラスについて説明します。それによって、このライブラリーのより大きな構造に対する俯瞰が出来るでしょう。後半部分では、モンテカルロ法や有限差分法モデルを構築する為に使われている、いくつかの特別なフレームワークについて説明します。その内のいくつかは、それ以外のものより、より洗練されています。このフレームワークの現時点での弱い部分は、強い部分と同じ位、興味深いものだと思います。

この本の読者としては、QuantLib ライブラリーを使って自ら金融商品や価格モデルを開発しようとしているユーザーを想定しています。もしあなたが、それに該当するなら、ライブラリーが提供するクラス階層やフレームワークに関する説明を読んで頂けると、ご自身のプログラムコードを QuantLib ライブラリーのそれと統合する為に必要な鈎について、何等かの情報を得られるでしょうし、提供されている機能を活用する助けになるでしょう。もし、そういった類の読者でないとしても、本を閉じないで下さい。そうで無い方にとっても、何等かの役に立つ情報を得られると思います。しかし、ここでひとつ、あるヒントを残します。別の QuantLib ライブラリーのアドミニストレーターの一人が、この本の姉妹版になる“Using QuantLib”を書くつもりであると繰り返し述べています。それが出来上がるまで、皆さんも、全力で彼を急かして下さい。

*　　*　　*

慣習に従って、この本についていくつかの注意点を述べておきます。

読者が C++ と金融工学に関する一定の知識がある事を前提にしています。私自身が、これらについて、この本で改めて教える事が出来るとは思っていません。この本は既に相当ぶ厚くなっており、その余裕もありません。ここでは、QuantLib ライブラリーの実装方法とデザインについての説明だけに留めます。金融工学における方程式が前提とする空間の説明や、C++ の文法や躓き易い点などの説明は、別のより詳しい方にお任せします。

読者の方は既に気づかれたかも知れませんが、書き手の主語に単数(“I”)を使っています。はい、確かに自己中心的ですが、それが不快な為に本を閉じないで下さい。しかし、もし書き手の主語を複数(“We”)としたら、それはそれで尊大です。書き手に第3者を使っても同じでしょう。色々考えた結果、私としては、あまり格式ばっていない居心地のいい方法を取る事にしました(従って、いろんな省略形も使っています)。同じ理由から、読者の方も、固い言い方の読者(”reader”)とせずにあなた(“you”)を使う事にします。単数の I を使うのは、混乱を避ける為でもあります。もし複数の We を使った場合、QuantLib ライブラリーの開発者全体を指すものと理解して下さい。

この本の中では、デザインの変遷についても、それ自体が興味を引くものであったり、最終形に影響するものであったりすれば、説明して行きます。論点を明確にする為、ほとんどの場合、行き止りや、道を間違えたケースは飛ばして、時と共に変遷していったデザインについては、一纏めにして、最終的に決定したデザインについてのみ、時には簡略化して、説明します。それでも、述

べないといけない事はたくさんあります。Alabama Shakes（アメリカのロックバンド。Boys and Girlsの歌詞参照）の言葉を借りれば、”why”には、恐るべき多くの数の質問が含まれています。

プログラムコードの説明の中で、そこで使われているデザインパターンについても説明します。注意しておきますが、読者が開発されるコードの中にそれを使うように勧めるものではありません。デザインパターンは、利用価値がある時にだけ使うべきであり、デザインパターンの為に使うべきではありません。（この点に関するより詳細な解説は巻末　Kerievsky, 2004 ご参照）しかし、QuantLib ライブラリーは何年にも渡るプログラミングとその修正の歴史を経ており、ユーザーからのフィードバックや新しい要求への対応をしてきました。その結果、デザインがデザインパターンに収束していくのは自然な事でもあります。

本の中で、プログラムコードの概要を示していますが、ライブラリーの中で使われているのと同じ作法（Appendix B 参照）を使っています。一つだけ、その作法からはずれている部分があります。行の長さの制約から、型名から std や boost の namespace を外している所があります。プログラムコードの Listing をチャートで補完する必要がある場合は、UML を使っています。UML に馴染みのない方は、その詳しいガイドは巻末Fowler, 2003 の文献を参照して下さい。

*　　*　　*

さて、イントロダクションは以上です。本編を始めましょう。

3. Financial instruments and pricing engines

「金融商品（の価格計算）ライブラリは、金融商品の価格を計算する方法を提供するものである」と言ったら、“ラ・パリサードの真理”と同じく、当たり前の事を述べているにすぎないかもしれません。しかし、ライブラリの設計者にとって、価格計算の方法の提供というのは課題の一部でしかありません。金融商品ライブラリには、将来、新たな価格計算機能を追加する為に、必要な機能を持たせる必要があります。（訳注：ラ・パリサードの真理とは、中世のフランス軍人 de La Palisse 氏の墓標の文が“もし彼が死んでいなければ、彼は生きていただろう”と誤訳された事から、当たり前の真実の事を意味する。）

機能拡張の要請は、2つの方向で必要になると予想されます。ひとつは、新しい金融商品への対応、もうひとつは、同じ商品について別の価格計算方法を追加していくことです。いずれの方向であっても、拡張性を持たせるには、いくつかの必要条件がありますが、デザインパターン的な用語で言えば、「いずれの方向の要請にも合致するものでなければならない」。本チャプターでは、そのような拡張性の要請とそれに対する解決方法について説明します。

3.1 The Instrument class

われわれ（金融モデルのプログラマー）の世界では、Financial Instruments は、それ自体ですでに基本概念です。従って、自らをオブジェクト指向プログラマーと自認している人なら、まずこれを Base Class として構成し、そこから様々な種類の金融商品の派生クラスを定義していくでしょう。

その考え方をもとにすれば、例えば（その Base Class が持つべき機能として）次のようなプログラムコードを書きたくなるのではないでしょうか

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

（訳注：ポートフォリオ配列の中に、どのような金融商品の派生クラスオブジェクトが含まれていても、ベースクラスで定義された共通のメソッド NPV( ) で価格が返ってくるような仕組みにしておけば、様々な商品のポートフォリオの総額を計算する場合、上記のような簡単なコードで記述できるという事）

このように関数を定義すれば、具体的な商品のタイプを気にする必要ありません。しかし、このプログラムコードでは NPV( ) のメソッドにどのような引数を渡せばよいのか解らないし、さらにどのようなメソッドを呼び出すのかも解りません。上記の一見無害に見える Code についても、一旦引いて、（この Base Class が持つべき）インターフェースについてよく考えてみる必要があります。

3.1.1 インターフェースと要件

金融市場では、—非常に単純な商品から非常に複雑な商品に至る—広範な商品が取引されており、ある特定の商品（例えば株式オプション）のクラスに特有のメソッドが、他の商品（例えば金利スワップ）では意味をなさないという事は十分起こり得ます。従って Instruments ベースクラスで定義すべき、すべての金融商品に共通のメソッドの数は極めて限られました。我々は、その共通するメソッドを、現在価値を返すメソッド（場合によっては推定誤差の値も含めて）と、その金融商品が満期を過ぎているかどうかを返すメソッドだけにしました。商品によって、（そのメソッドが）どのような引数を取るべきかについて、（最近改変された C++11の variadic templates のような技術を使っても）、ベースクラスでは特定できない為、そのメソッドは引数を取らない形で宣言しています。従って、メソッドに必要な入力項目は、金融商品ごとに、保存されなければなりません。その結果、Instrument クラスのインターフェースは下記 Listing2.1のようになっています。

Listing 2.1: Preliminary interface of the `Instrument` class.

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

プログラミングの慣行に従って、当初、各メソッドはすべて純粋仮想関数にしました。ただ、これは（ガーシュインのオペラ Porgy and Bess に登場する麻薬密売人の Sportin' Life が指摘したように）必ずそうしなければならないというものではありません。ベースクラスで具体的に定義・実装した方が良いようなプログラムの動作があるかもしれません。本当にそうなのか調べるために、一般的な金融商品に期待される共通の動作が何であるのか、またそれを実行させる為のプログラムで共通するところはあるのか、よく分析してみる必要がありました。その結果、QuantLib の開発過程のなかで、2つの必要事項が見つかり、開発期間中に実装の方法を変更しました。ここでは、それらの現時点での実装内容を説明します。

ひとつは、特定の金融商品について、価格計算の方法が複数存在し（例えば、解析解で価格計算をする方法と、数値解でそれを行う方法）、その実装を継承の仕組みを使わずにできないだろうかということです。読者の方はこれを聞いて、Strategy Pattern を使えばいいのではと思われるかもしれません。その通りです。本チャプターの Section2.2で、その実装方法を説明します。

もうひとつは、金融商品の価格は、市場データに依存しているという事実からくるものです。市場データは、本質的に時間の経過によって変化するものです。従って、その商品の価格も時間の経過によって変化していきます。また、金融商品の価格変化は、市場データを提供するソースが複数存在しうるという点からも発生します。我々は、ひとつの金融商品が複数のデータソースとリンクできるようにする必要があると考えました。その場合、価格計算のメソッドを呼び出した時は、その都度、最新データを使って金融商品の価格を再計算させるような仕組みが必要と考えました。また、複数のデータソースを任意に選択できるような仕組みも必要と考えました。そして、デー

タソースを変更した場合は、その金融商品にとっては、市場データのアップデートと同じ取扱い（都度価格の再計算を行う）とすべきとも考えました。

効率性の低下も関心事でした。例えば、多くの金融商品をひとまとめにして、定期的に時価を確認して、ポートフォリオの全体の価値を適宜モニターするとします。その際、単純な仕組みでは、時価の変更が無かった商品についても、すべて価格の再計算を行ってしまう可能性がありました。従って、Instrument クラスのメソッドの中に、時価を一時保管する仕組みを取り入れました。時価の再計算は、市場データの更新があった場合のみ行い、それ以外の場合は、一時保管されている値を使うような仕組みです。

3.1.2 実装

どの金融商品についても共通な、価格を一時保管したり、再計算したりする仕組みを司るプログラムコードは、2種類のデザインパターンを使って書かれています。（デザインパターンについては巻末(Gamma *et al*, 1995) の文献を参照）

入力データに変更があった場合、Instrument クラスは Observer パターンを使って、その変更の通知を受ける仕組みにしています。Observer パターンについては、Appendix A を参照下さい。（但し、Appendix を読んだからといって、巻末(Gamma *et al*, 1995) にある「Gang of Four(4人組)」の（有名な）本を読まない理由にしないでください。）とりあえず、ここではどのようにそれが使われているか解説します。

（金融商品と市場データとの関係から）明らかなように、Instrument クラスは Observer の役割を持ち、市場データは Observable の役割をもっています。入力データ（訳注：市場データ）が更新された場合、それを（Observer である Instrument に）通知されるような仕組みを構築する為には、Observer が、その入力データを保管しているオブジェクト（Observable）への‘参照’を保持しておく必要があります。このような仕組みにするには、何らかの Smart Pointer（訳注：メモリーリークを防ぐため、ポインターの指し示すメモリー領域の管理を自動的に行ってくれるポインター）が必要になると気づかれるでしょう。しかし、ポインターの仕組みだけでは、我々が必要としているメカニズムにとって十分ではありません。すでに説明した通り、入力データの変更は、市場価格が動く以外に、データソースの変更からも発生します。（訳注：例えば、同じ銘柄の株が複数の取引所で取引されている場合、データソースは一本だけではない）

（Smart）ポインターを1つしか持たない場合、それが参照するオブジェクトの直近のデータのみにしかアクセスできません。我々が必要とするのは、ポインターを保持するポインターです。この仕組みは、QuantLib の中では、Handle という名前のクラス・テンプレートを使って実装されています。これについても、詳細は Appendix A を参照下さい。このセクションの論点との関連で言えば、ある特定の Handle の複数のコピーは、ひとつのオブジェクトに対するリンクを共有しているという点です。仮に、ポインターが別のオブジェクトを指すようになった場合、Handle のすべてのコピーにそれが通知され、それらの Handle を保持しているオブジェクト（訳注：Instruments）に対し、新しいポイント先（のデータ）を通知する仕組みになっています。さらに、Handle は、そこが指し示しているオブジェクト（訳注：市場データを保持するオブジェクト）から、それを監視するすべての Observer（訳注：市場データをモニターしている Instruments）に対し、データ変更の通知を転送する仕組みを持っています。

そして、市場データの役割を持ち、Handle クラスの中で保持できる、様々な種類の Observable クラスを実装しました。その中で、最も基本的なクラスが Quote クラスで、市場価格を一個だけ保持する機能を持っています。金融商品の価格計算に必要な市場データ（Observable）は、その他にも、もっと複雑な構造を持つイールドカーブやボラティリティの TermStructure クラスなどがあります。（それらのクラスも、最終的には一個しかデータを持たない Quote クラスのインスタンスに行きつきます。イールドカーブオブジェクトは Bootstrapping を行うため、預金金利やスワップレートを保持する Quote クラスのインスタンスを必要とします）

Instrument ベースクラスが持つもう1つの課題は、具体的な価格計算のアルゴリズムは派生クラスで実装するとして、ベースクラスにおいて価格の再計算を行った後の値を一旦保持しておくメカニズムを、どのように抽象化するか、という事です。このメカニズムは、Template Method パターンを使って導入しました（このパターンについても巻末 [1] の本を参照）。QuantLib の初期の頃のバージョンでは、この仕組みを Instrument クラスそのものに、含めていました。その後のバージョンで設計を変更し、その仕組みを抜き出して別のクラスに持たせることとしました。そのクラスは LazyObject という名前で呼ばれており、ライブラリの中で、他の用途にも使われています。このクラスの概要は、下記 Listing 2.2に示します。

Listing 2.2: Outline of the `LazyObject` class.

```
class LazyObject : public virtual Observer,
                   public virtual Observable {
  protected:
    mutable bool calculated_;
    virtual void performCalculations() const = 0;
  public:
    void update() { calculated_ = false; }
    virtual void calculate() const {
        if (!calculated_) {
            calculated_ = true;
            try {
                performCalculations();
            } catch (...) {
                calculated_ = false;
                throw;
            }
        }
    }
};
```

プログラムコードは極めて単純です。bool 型のメンバー変数 `calculated_`は、価格の計算結果が“直近のものかどうか”のフラッグを保持します。ここにある `update()` メソッドは、（ベースクラスである）Observer のメソッドを実装したものであり、Observable クラスから（入力データが更新されたという）通知が届いた場合に、メンバー変数 `calculated_`を false にセットし、すでにある価格計算結果を、古いものとして扱います。

`calculate()` メソッドは、Template Method パターンを使って実装されています。巻末(Gamma *et al*, 1995) の4人組の本にある通り、メソッドの内容のうち、すべての派生クラスに渡って共通な動作はベースクラスで実装され（このケースでは一時保存された計算結果のフラッグの取扱い）、派生クラス毎に実装すべき部分については、仮想関数（このケースでは `performCalculations()`）に任されます。その仮想関数は、ベースクラスのメソッドの中で呼び出されます。従って、派生クラスでは、具体的な計算方法を（`performCalculations()` の実装の中で）特定すればよいだけで、計算結果の一時保存について考える必要がありません。派生クラスの実装部分は、ベースクラスのメソッドの中に（コンパイルの際に）組み込まれます。

計算結果の一時保存のロジックは単純です。もし、現在保存されている計算結果データがすでに古くなってしまった場合は（`calculated_` フラッグが false にセットされている）、派生クラスにおいて価格の再計算を行い、`calculated_`フラッグを再び true に戻します。もし現在保存されているデータが最新分であれば何もしません。

ロジックは簡単でも、実装はそれほど単純ではありません。まず、bool 型である `calculated_`のフラッグ変更の手続きを、なぜ try ブロックの手前で行う必要があったのでしょうか？また、catch ブロックの中で、なぜ `calculated_`フラッグの設定を元に戻す操作を行ってからエラー処理へ飛ぶのでしょうか？　もしかしたら、以下のようなプログラムコードであっても、同じ効果を持たせる事ができたのではないか、と考えられませんか？

```
if (!calculated_) {
    performCalculations();
    calculated_ = true;
}
```

こうしなかった理由は、`performCalculations()` メソッドが再帰的に `calculate()` メソッドを呼び出す場合が想定されるからです（例えば LazyObject がイールド TermStructure で、Bootstrapping を行うような場合）。もし `calculated_`を true にセットしなかった場合、上記構文中の if 条件は継続してチェックされ、`performCalculations()` が再び呼びだされ、無限ループに陥ってしまいます。（`performCalculations()` が実行される前に）このフラッグを true にセットする事により、そういった事が防げます。しかし、（仮に `performCalculations()` の実行時に）エラーにより例外処理に飛ぶ場合は、その前に、このフラッグを false に戻しておく必要があります。その後、さらに再例外処理に飛び、インストールされているエラー処理 Handler にプロセスが移動します。

LazyObject には、さらにいくつかのメソッドが用意されており、ユーザーに再計算を行わせないようにしたり、逆に強制したりできるようになっています。これについてはここでは説明しません。興味がある方は、かの有名な Obi-Wan Kenobi のアドバイスに従って下さい —“Read the Source, Luke”。

Instrument クラスは LazyObject から派生しています。上で説明した LazyObject のインターフェース（ここでは `calculate()` メソッド）を、Instrument クラス用にデコレート（訳注：Decorator Pattern を使って、ベースクラスの仮想関数 `calculate()` に、派生クラスで機能を追加している）しています。その内容は、若干の追加のコードも含めて、下記 Listing 2.3のとおりです。

Listing 2.3: Excerpt of the `Instrument` class.

```
class Instrument : public LazyObject {
  protected:
    mutable Real NPV_;
  public:
    Real NPV() const {
        calculate();
        return NPV_;
    }
    void calculate() const {
        if (isExpired()) {
            setupExpired();
            calculated_ = true;
        } else {
            LazyObject::calculate();
        }
    }
    virtual void setupExpired() const {
        NPV_ = 0.0;
    }
};
```

ここで追加されたコードも、Template Method パターンに従って、派生クラスに、その Instrument に固有な計算アルゴリズムの実装を委任できるようにしています。(訳注:このクラスの段階で `performCalculations()` を実装しておらず、さらに派生するクラスでの実装を前提にしている。)このクラスは、価格計算結果を一時保存するメンバー変数、NPV_を持っています。派生クラスではそこで必要なメンバー変数を追加で宣言する事ができます。(注:Instrument クラスでは、ここでは書いていませんが、その他に errorEstimate_変数が定義されており、NPV_に関する論点がそのまま適用できます)。

`calculate()` メソッドの本体は、仮想関数である `isExpired()` を呼び出して、その商品の期限が到来しているかどうかをチェックしています。かりに期限が到来しているのであれば、さらに仮想関数 `setupExpired()` を呼び出し、NPV_を適切な値にセットします。デフォールトの設定は NPV_を 0に設定しますが、派生クラスで別の適切な値とする事も可能です。その後 `calculated_`フラッグを true に設定します。もし、その商品がまだ満期を迎えていないのであれば、ベースクラスである LazyObject の `calculate()` メソッドが呼び出され、そのメソッドはさらに `performCalculations()` を呼びだします。このような仕組みをとると、`performCalculations()` に一定の作業を必ずさせる必要があります。すなわち NPV_に計算結果を、(メンバー変数に)書き込む事です。(その他派生クラスに特有のメンバー変数への値の代入も必要になります。)最後に商品の価格(NPV)を返す NPV( ) メソッドですが、NPV_の値を返す前に、`calculate()` メソッドを呼び出して、その値をまず計算するようにしています。

Aside: const にすべきか否か？

変数 `NPV_`が、なぜ `mutable` で宣言されているかの説明に興味を持たれるかも知れません。なぜなら、データの一時保存や lazy calculation を実装する際には、この問題が常に発生するからです。問題の要点は、`NPV()` メソッドは、論理的には `const` メソッドであり、商品の価格計算は、データを変更しないという事です。従って、Library のユーザーは、そのようなメソッドを `const` なインスタンスから呼び出しても、意図せざるデータ変更が為される事はないという事を期待してかまいません。その代わり、NPV( ) メソッドが `const` で宣言されているという事は、`calculate()` や `performCalculations()` などのメソッドも `const` で宣言しなければなりません。しかし、我々が目的とするところの、価格計算を lazily に行ったり、計算結果を一時保存したりする機能を実装しようとすると、そういったメソッドの本体中にデータ書き換えが可能な変数を必要とします。データを一時保存する為のメンバー変数を `mutable` として宣言することによりそういった問題が解決されます。こうする事によって、我々開発者やのちの派生クラスの実装を行う開発者にとって、両方の目的、すなわち `NPV()` メソッドの `const` 化と、メンバー変数の lazy な役割を担わせる事、が達せられます。

また、C++ 11では、`const` メソッドをマルチスレッドでも安全性を持たせる必要がある点を注意しなければなりません。すなわち、二つのスレッドが同時に const なメンバーメソッドを呼び出す場合、競合しないようにしなければなりません（詳しくは巻末(Sutter, 2013) の Sutter の著書を参照）。新しい C++ のスタンダードに準拠するために、mutable として宣言されたメンバー変数を mutex で保護しなければなりません。いずれ、デザインの変更が必要になってくると思われます。

3.1.3 例：金利スワップ

このセクションの説明の最後に、これまで説明してきた（Instrument クラスの）機能をもとに、個別の金融商品をどのように実装するか説明します。

例として‘金利スワップ’を使いたいと思います。当然ご存知の通り、この商品は、定期的にキャッシュフローを交換する契約です。この商品の価値（Net Present Value ‘NPV’）は、受取キャッシュフローと支払いキャッシュフローの現在価値を計算し、合算したものです。

当然ながら、金利スワップは Instrument クラスの派生クラスとして実装されています。その概要を Listing 2.4に示します。（この実装内容は、若干古いバージョンのものですが、簡単に説明できるので、例として使います）

Listing 2.4: Partial interface of the `Swap` class.

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

このクラスは、メンバー変数として、価格計算に必要な情報のオブジェクトを持っています。すなわち、2つのキャッシュフロー（Cashflow Legs）と、それを現在価値に割引くための金利期間構造（Yield Term Structure）です。それに加え、計算結果を格納するための2つの変数を持ちます。さらに、（ベースクラスである）Instrument クラスのインターフェースを、具体的に実装する為のメソッドと、それ以外にも金利スワップに特有の値を返すメソッドを宣言しています。（訳注：`isExpired()`、`setupExpired()`、`performCalculations()`、はベースクラスで仮想関数として宣言されている。また `firstLegBPS()` 、`secondLegBPS()`、は Swap に特有の値を返すメソッド）
Swap クラスとそれに関連するクラスの関係図を、下記 Figure 2.1に載せておきます。

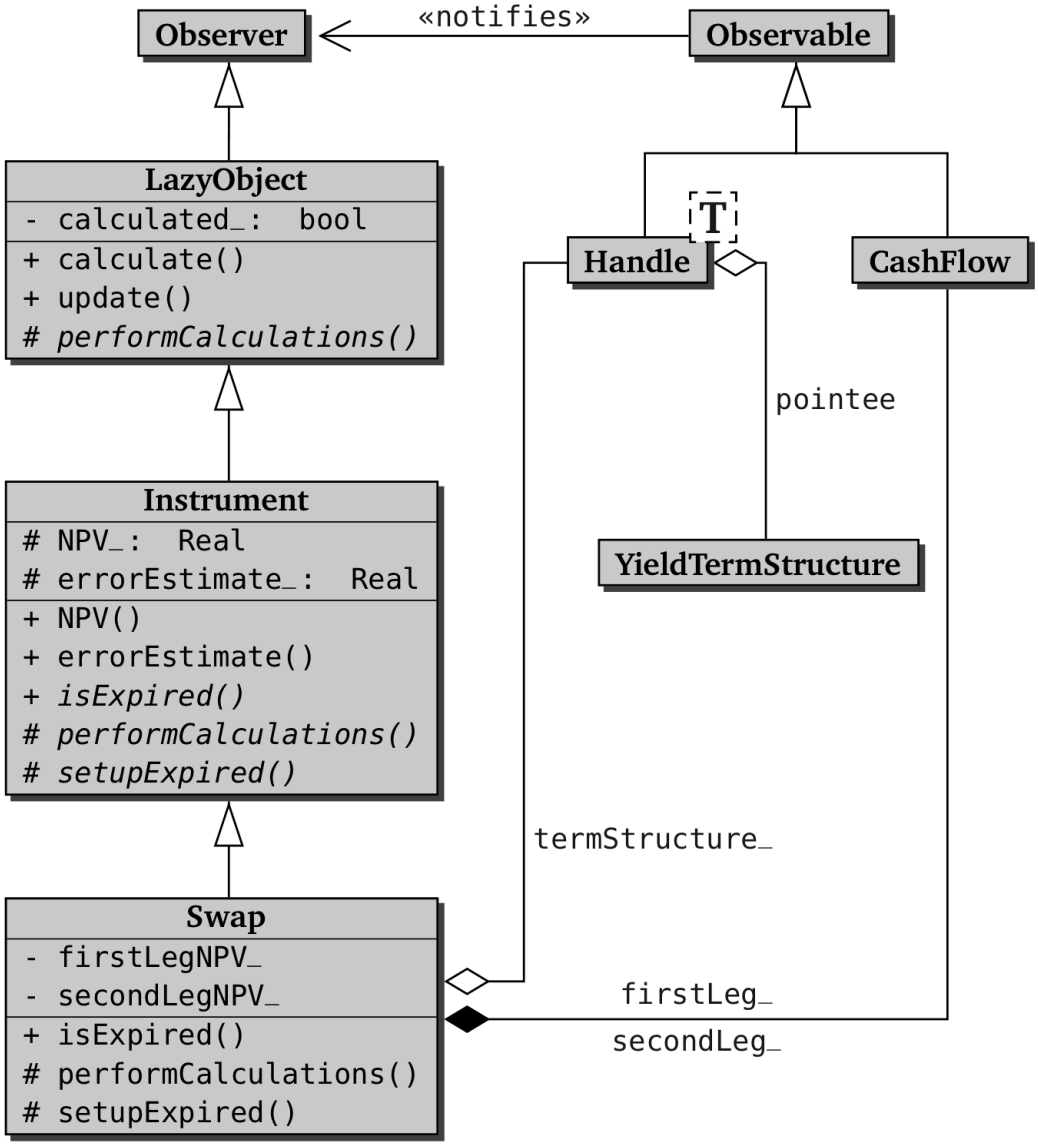

Figure 2.1. Class diagram of the `Swap` class.

このクラスを Instrument ベースクラスへ適合させる為、3つの手順を踏んでいますが、3番目の手順は派生クラスによっては任意です。それに関係するメソッドを下記 Listing に示します。

Listing 2.4: Partial implementation of the `Swap` class.（続き）

```
Swap::Swap(const vector<shared_ptr<CashFlow> >& firstLeg,
           const vector<shared_ptr<CashFlow> >& secondLeg,
           const Handle<YieldTermStructure>& termStructure)
: firstLeg_(firstLeg), secondLeg_(secondLeg),
  termStructure_(termStructure) {
    registerWith(termStructure_);
    vector<shared_ptr<CashFlow> >::iterator i;
    for (i = firstLeg_.begin(); i!= firstLeg_.end(); ++i)
        registerWith(*i);
    for (i = secondLeg_.begin(); i!= secondLeg_.end(); ++i)
        registerWith(*i);
}

bool Swap::isExpired() const {
    Date settlement = termStructure_->referenceDate();
    vector<shared_ptr<CashFlow> >::const_iterator i;
    for (i = firstLeg_.begin(); i!= firstLeg_.end(); ++i)
        if (!(*i)->hasOccurred(settlement))
            return false;
    for (i = secondLeg_.begin(); i!= secondLeg_.end(); ++i)
        if (!(*i)->hasOccurred(settlement))
            return false;
    return true;
}

void Swap::setupExpired() const {
    Instrument::setupExpired();
    firstLegBPS_= secondLegBPS_ = 0.0;
}

void Swap::performCalculations() const {
    NPV_ = - Cashflows::npv(firstLeg_,**termStructure_)
           + Cashflows::npv(secondLeg_,**termStructure_);
    errorEstimate_ = Null<Real>();

    firstLegBPS_ = - Cashflows::bps(firstLeg_, **termStructure_);
    secondLegBPS_ = Cashflows::bps(secondLeg_, **termStructure_);
}

Real Swap::firstLegBPS() const {
    calculate();
    return firstLegBPS_;
```

```
}

Real Swap::secondLegBPS() const {
    calculate();
    return secondLegBPS_;
}
```

最初のステップは、コンストラクターによって実行されており、2種類の Cash Flows オブジェクトと、それらを現在価値に割引く為に必要な、金利の Term Structure オブジェクトを引数で取り、それぞれメンバー変数へ代入します。さらにコンストラクターによって生成された Swap オブジェクト自身を、Cash Flows オブジェクトと Term Structure オブジェクトに対し、Observer として登録する事も含まれています。前のセクションでも説明した通り、この登録により各 Cash Flow オブジェクトは、その内容に変更があった場合は、(Observer である)Swap オブジェクトに通知し、価格の再計算を促します。

2番目のステップは、必要なインターフェース(各種メソッド)を実装する事です。`isExpired()` メソッドのロジックは極めて簡単で、メンバー変数に保持しているキャッシュフローについて、それぞれの支払日をチェックします。ひとつでも、支払日が到来していないキャッシュフローが存在していれば、not expired として情報を返します。もし未到来のキャッシュフローが全く無い場合は、その商品は expired となります。その場合、`setupExpired()` メソッドが呼び出されます。このメソッドはベースクラスである Instrument クラスの持つインターフェースですが、ここでは、そのベースクラスのメソッドを呼び出すのに加え、各 Cash Flows Leg の現在価値を0に設定する動作をします。

最後のステップは、`performCalculations()` の実装です。この計算は、外部の Cashflows クラスが持つ2つの関数を呼び出して行われます。(読者の方がもし何か騙されていると感じたならば、ここでの具体例のポイントは、計算手順をどのように実装するかではなく、それをひとつのクラスの中にまとめる方法を解説するのが目的であるという点を考慮して下さい。おかしいと思われる点については、後の章で、Cashflows クラスとそれに関連する複数の関数の説明を行いますので、そこで確認して下さい。)　ひとつめの呼び出される関数は、npv( ) で、上記で概略を説明したアルゴリズムを、そのまま実装したもので、将来キャッシュフローの現在価値を順番に足しあげています。変数 NPV_には、2つのキャッシュフローの現在価値の差額(ネット金額)を代入します。ふたつ目の外部関数は、`bps()` で、両方のキャッシュフロー全体の basis-point sensitivity(金利感応度)を計算しています。キャッシュフロー Leg 毎に関数が呼び出され、計算結果はそれぞれ別のメンバー変数に格納されます。計算結果についは(数値解析による価格計算を行っていないので)推定誤差がゼロであり、`errorEstimate_`変数は　`Null<Real>()` ー特別な浮動小数点の値で、無効な数字を表示するために使用されるーに設定されます。(NaN (Not a Number) を使う方が良かったかもしれませんが、それを感知する方法を異なる環境へ移植するのが困難です。Boost::optional を使う方法については調べてみる必要があるかも知れません)

最後のステップは、この派生クラスで追加された、計算結果を保持するメンバー変数の値を出力する手順です。ここでは、その変数に対応するメソッド(`firstLegBPS()` と `secondLegBPS()`)が実装されており、その保持された変数を出力する際、まず lazily に(訳注:入力データの変動があった場合にのみ実行されるという意味)計算を実行するようにしています。

Swap クラスの実装は以上です。Instrument クラスに被せてプログラムコードを書くことにより、

Swap クラスでは親クラスのプログラムコードの恩恵を受けています。すなわち、入力データの変更通知があったかどうかによって、価格の再計算を行い、その値を一時保存するという動作を、Swap クラスでプログラムコードを書かなくても、行わせることができるようになっています。

Aside: Handle と共有ポインター

Swap クラスのコンストラクターが、引数として、Discount Curve を Handle（ハンドル：前の章で説明した、ポインターに対する Smart（賢い）ポインター）で受取り、キャッシュフローを単純な共有ポインターで受け取っている事について疑問に思われるかもしれません。その理由は、Discount Curve については、（価格計算において）別のカーブを使ってみる可能性があるのに対し（従って、それが可能な Handle を使う）、キャッシュフローについては、Swap の定義（Swap クラスの個別のインスタンス）の重要な一部であり、それを変更する事は想定されない為です。

3.1.4 今後の開発予定

読者の方は、上の具体例と Instrument クラス全般について、私が定義した方法に欠点があることに気付かれたかもしれません。汎用化したにもかかわらず、我々が実装した Swap クラスは、2つのキャッシュフローが異なった通貨の場合、対応できません。Instrument クラスについても、もしユーザーが通貨の異なる2種類の Instrument の価値を合算しようとする場合にも同様の問題が発生します。その場合、ユーザーの方はマニュアルで、自分で一方の通貨の価格をもう一方の商品の通貨の価格に換算してから行う必要がありあす。

このような問題は、プログラムの実装におけるひとつの弱点から発生しています。すなわち、価格計算の結果を Real 型（単純な浮動小数点の型）で返すようにしてしまった事です。その結果、計算結果について、実務の世界では当たり前の、通貨の情報が失われてしまいました。

この弱点については、計算結果を（Real ではなく）Money クラスで返すようにすれば解消するかもしれません。このクラスのインスタンスは通貨の情報を持たせることができ、さらに、ユーザーのセッティング次第では、自動的に共通の通貨（例えば、会計で使う通貨）に換算する動作を加える事も可能です。

しかし、これは非常に大きな変更になり、プログラムコードの相当大きな部分に、複雑な形で影響を与えることになります。従って、この弱点への対応には、相当真剣に取り組まなければなりません。

もうひとつの（より微妙な）弱点は、Swap クラスは2種類の抽象化された Components（訳注：インスタンスの持つ基本情報のオブジェクト）を明確に分離できていない事です。即ち、スワップ契約を特定する情報（Cash Flows）と、それを時価評価するための市場データ（Term Structure）を明確に区分できていない事です。

その問題の解決方法は、Instrument のインスタンスの中に、前者の情報（タームシートに書かれているスワップ契約の内容そのもの）のみを保持させ、市場データについては別の所で持たせるようにする事です。（概念的にその方がより明確である事に加え、外部の関数が、その金融商

品の情報を別のフォーマットに変換するような場合ー例えば、FpML フォーマットに変換したり、あるいは戻したりするような場合ーに、非常に有効になるでしょう。)(訳注:FpML:Financial Products Markup Language: XML をベースにしたデリバティブズ取引などの情報交換の為の言語)

その方法については、次のセクションで説明します。

3.2 価格計算エンジン

この Chapter の冒頭で申し上げた、Model Library に要求される2つの要請の内、2つ目について説明します。ある具体的な金融商品について、価格計算の方法は、1つとは限らない事、さらにユーザーは様々な理由で複数の計算方法を使う場合がある、という事です。ひとつ典型的な例として、株のヨーロピアンオプションで考えてみましょう。ある人は、Black-Scholes の公式を使って、オプションの市場価格から Implied Volatility(オプション価格に内包されている Volatility)を計算したいと考えます。あるいは、そこから、Stochastic Volatility Model のカリブレーション(パラメータの調整)を行って、より複雑なオプションの価格計算に使ったりします。さらに、Black-Scholes の公式と Finite-difference scheme(有限差分法)を使ったアルゴリズムの計算結果を比べて、有限差分モデルの検証に使ったりします。さらには、モンテカルロシミュレーション法による複雑なオプション価格の計算において、ヨーロピアンオプションの解析解を、control variate(訳注:モンテカルロシミュレーションによる解の収束速度をアップさせる為の制御変量)として使ったりします。

従って、ひとつの金融商品について、複数の方法で価格計算ができるような仕組みにしたい訳です。その場合、当然ながら `performCalculations()` メソッドを複数個実装するという方法は、望ましくありません。なぜなら、そうする為には、ひとつの金融商品について、複数のクラスを作らなければならなくなるからです。そうする為には例えば、EuropeanOption クラスから、AnalyticEuropeanOption クラスと McEuropeanOption クラスを派生させる事になりますが、この方法だと、2つの点で問題があります。まず概念的に考えて、ひとつの商品の為にふたつの箱を用意するようなものです。Gertrude Stein の言い回しを使えば、「ヨーロピアンオプションがヨーロピアンオプションであると言っているのは、ヨーロピアンオプションはヨーロピアンオプションであるという意味である。」という事です(訳注:Stein のオリジナルの表現は、"Rose is a Rose is a Rose is a Rose.")また、使用上の問題として、run-time(プログラム実行時)に価格計算方法の変更ができないという点です。

この問題の解決方法は、Strategy Pattern を使う事です。すなわち、その商品のオブジェクトに、価格計算方法をカプセル化した他のオブジェクトを持たせる事です。そのようなオブジェクトを Pricing Engine(価格計算エンジン)と呼ぶことにします。ある特定の商品オブジェクトが、その商品に対応した Pricing Engine を選択できるようにし、その選択した Pricing Engine に、計算に必要な引数を渡し、その Engine が価格計算を行い、その Engine から計算結果を返させるような仕組みが想定されます。そうすると、Instrument オブジェクトのインターフェースである `performCalculations()` の実装は、おおむね下記のようなプログラムコードになると推察されます。

```
void SomeInstrument::performCalculations() const {
    NPV_ = engine_->calculate(arg1, arg2, ... , argN);
}
```

ここでは、(performCalculations() の中で呼び出される)Pricing Engine の calculate() メソッドを、ベースクラスで仮想関数としてインターフェースを定義し、具体的な計算方法は、派生クラスとなる Pricing Engine で実装する事が想定されます。

残念ながら、この方法によるアプローチは、想定通りに機能しません。問題は、calculate() を呼び出すプログラムコード(訳注:performCalculations() の実装部分)を、ベースクラスの中で1回の実装で済ませたいのに、それが出来ないという事です。なぜならベースクラスは、個別の商品毎に必要な、Pricing Engine の引数を知る余地がありません。それらは Instrument の派生クラス毎に、数も型も千差万別です。同じ事は、Pricing Engine が返す計算結果についても言えます。例えば、金利スワップであれば、固定金利と変動金利スプレッドの Fair Value(市場レートにフィットした固定金利の水準)を計算して返す事が想定されますが、一般的なヨーロピアンオプションでは、価格以外に様々な Greeks(訳注:ギリシャ文字、転じて様々なリスク感応度のこと。)を返す事も想定されます。

上記のサンプルコードように、Pricing Engine に渡す引数を明示して列挙するようなインターフェースは、望ましくない結果をもたらします。異なる金融商品に対応する、異なる Pricing Engine は、当然ながら異なったインターフェースを持つはずであり、それを一本のベースクラスで表現する事はできません。そうすると、Instrument から Pricing Engine の calculate() を呼び出すメソッドは、Instrument の派生クラス毎に作り直さなければならなくなります。そこに大変な困難が存在します。

我々が選択した解決策は、Pricing Engine の引数や返値を、arguments と results という柔軟な構造体でやりとりする方法でした。この2つの構造体を派生させて、個別の金融商品(の Pricing Engine)ごとに必要な“引数”と“返し値”の構造体を作り、Pricing Engine オブジェクトに保持させます。また Instrument(の派生クラスの)オブジェクトは、Pricing Engine の持つ引数と返し値の情報を、書き込んだり読み込んだりできるようにします。

Listing 2.5に、このような構造を組みこんだ Pricing Engine クラスの定義と、その内部クラスとして定義された arguments クラスと results クラス、さらに、補助的役割を持つ GenericEngine クラス・テンプレートのコードを示します。この最後のクラスは、Pricing Engine のインターフェースの大半を実装しており、QuantLib を利用したい開発者は、個別の Pricing Engine の calculate() メソッドのみを実装すれば良いだけです。arguments と results クラスには、使いやすいように(入力データの)ドロップ Box を使えるメソッドが備え付けられています。すなわち arguments::validate() メソッドは、入力データとして受け取った引数が、有効なデータ範囲の中に納まっているかどうかチェックする機能をもっています。また、results::reset() メソッドは、Pricing Engine が価格計算を開始する前に、前回計算した結果を消去します。

Listing 2.5: Interface of `PricingEngine` and of related classes.

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

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

class PricingEngine::results {
  public:
    virtual ~results() {}
    virtual void reset() = 0;
};

// ArgumentsType must inherit from arguments;
// ResultType from results.
template<class ArgumentsType, class ResultsType>
class GenericEngine : public PricingEngine {
  public:
    PricingEngine::arguments* getArguments() const {
        return &arguments_;
    }
    const PricingEngine::results* getResults() const {
        return &results_;
    }
    void reset() const { results_.reset(); }
  protected:
    mutable ArgumentsType arguments_;
    mutable ResultsType results_;
};
```

この考え方をベースに作られた新しいクラスを使って、汎用的な `performCalculations()` の実

装例を示すことができます。すでに述べた Strategy パターンに加え、ここでは Template Method パターンも使って、Instrument の派生クラスで足りない機能を書き加えられるようにしています。そのプログラムコードを Listing 2.6に示します。インナークラスの `Instrument::results` クラスが定義されている所に注意して下さい。このクラスは、`Pricing Engine::results` クラスから派生しており、個別の Instrument 用の計算結果を保持します。(`Instrument;;results` クラスは、`std::map` オブジェクトをメンバー変数として保持しており、そこに追加の計算結果を保持できるようになっています。但し、それを記述したコードは下記では省略されています。)

Listing 2.6: Excerpt of the `Instrument` class.

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

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

```
        }
        Real value;
        Real errorEstimate;
    };
```

`performCalculations()` メソッドの実際の動作は、それを補助する複数のクラス（すなわち、Instrument, Pricing Engine、arguments および results の各クラス）に作業分担されています。そのような補助の動作（次の段落で説明）を理解するには、下記の Figure 2.2：UML シーケンス図（Unified Model Language Sequence）をよく見て下さい。さらに、各クラス間の静的な関係は Figure 2.3を参照下さい。

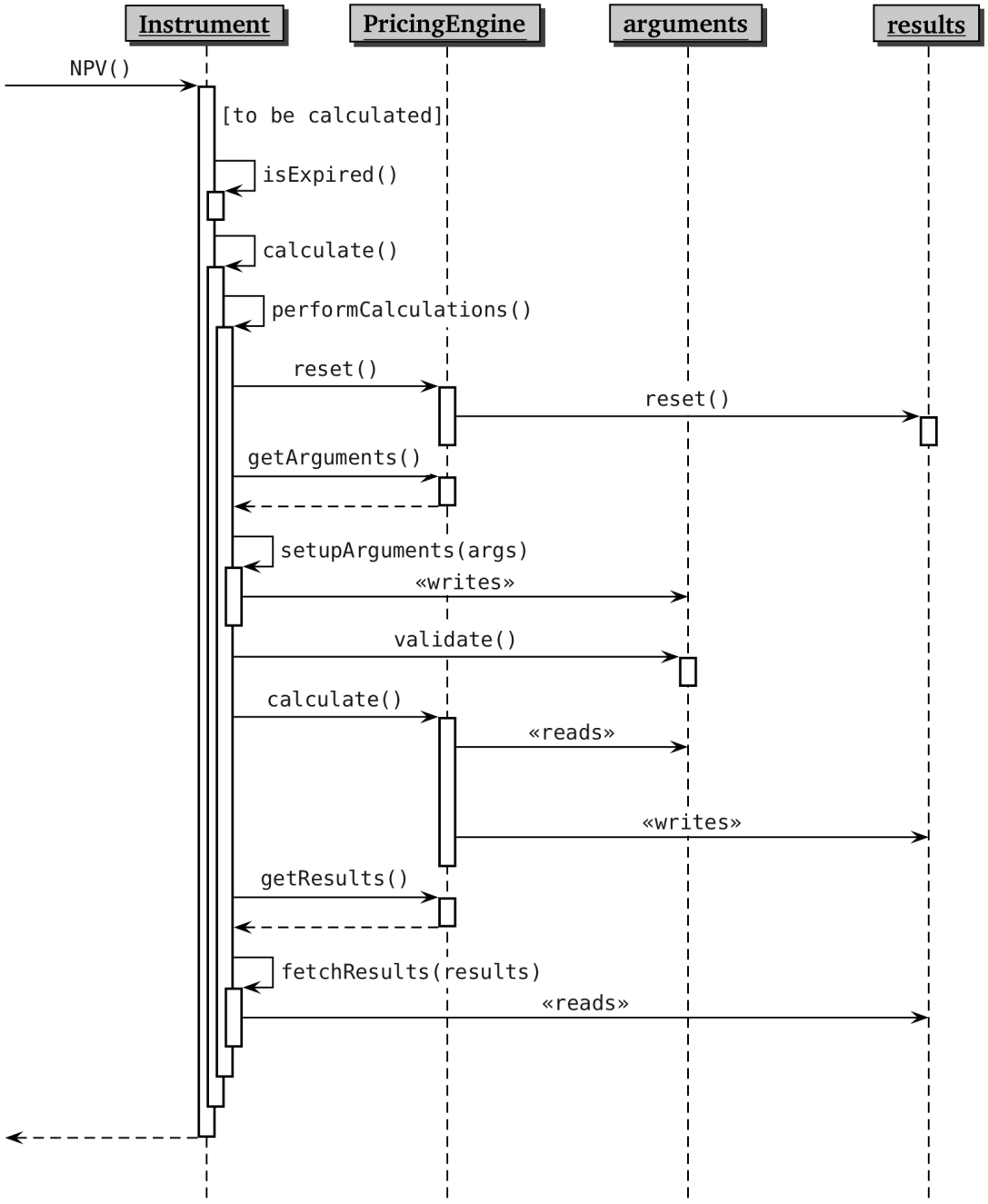

Figure 2.2. Sequence diagram of the interplay between instruments and pricing engines.

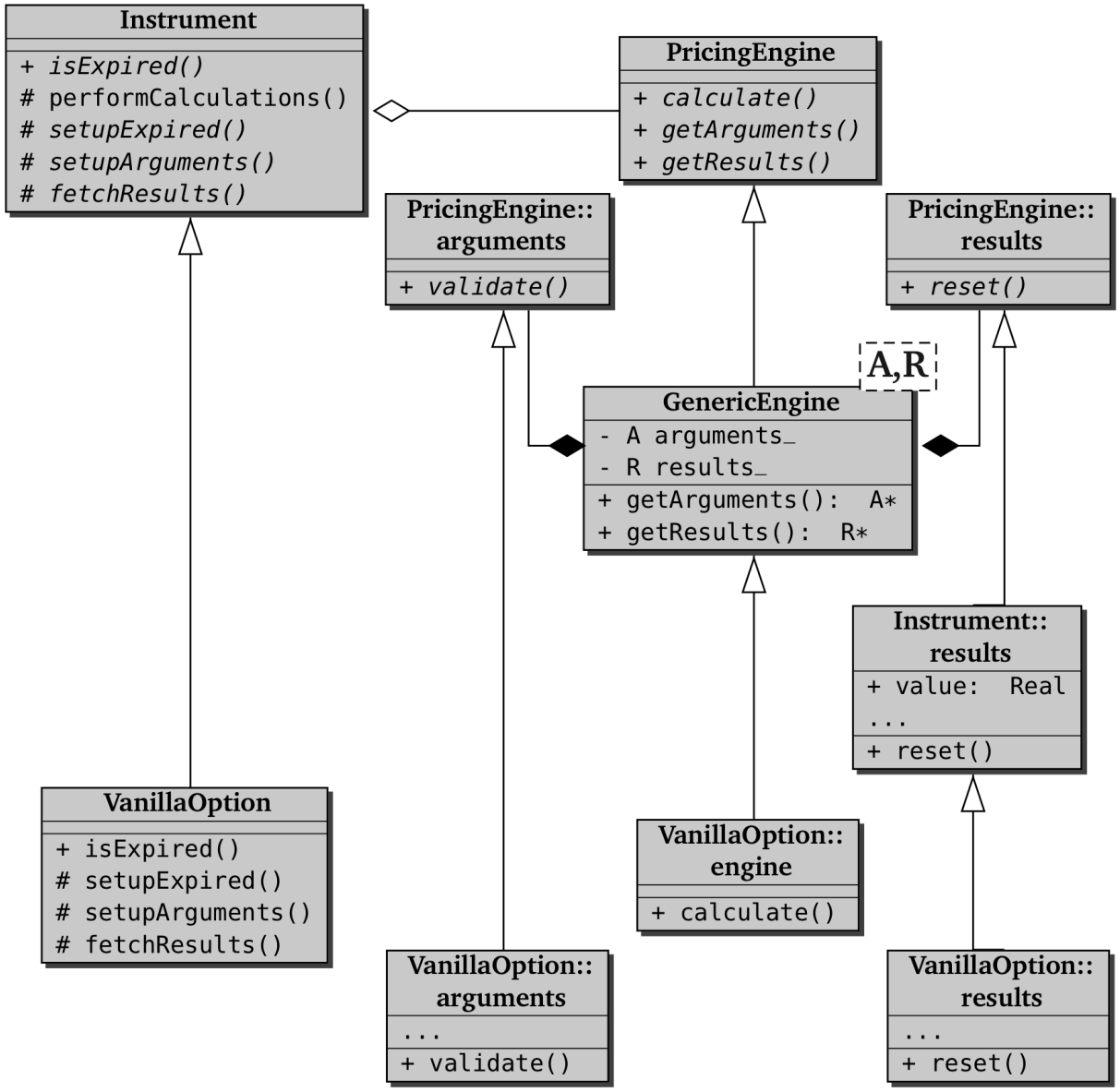

Figure 2.3. Class diagram of `Instrument`, `PricingEngine`, and related classes including a derived instrument class.

Instrument クラスの NPV( ) メソッドを呼び出すと（もしその商品が満期を迎えていなくて、価格やリスク量を計算する必要がある状態にあるのなら）、そこからさらに `performCalculations()` を呼び出します。そこから Instrument オブジェクトと Pricing Engine との間の連携が始まります。まず最初に、Instrument は、Pricing Engine そのものが存在しているかどうかチェックし、もしそうでないなら計算をそこでストップします。もし Pricing Engine が見つかったなら、Instrument は、Pricing Engine に対し、データのリセットを指令します。その指令は、reset( ) メソッドを使って、その Instrument 固有の results 構造体に伝達され、その構造体の中の古いデータを消去し、新しい計算結果を保持できる状態にします。

ここで、Template Method パターンが登場します。Instrument オブジェクトは、Pricing Engine オブジェクトに対し、arguments 構造体を渡すよう要求し、Pricing Engine は、その構造体のポインターを返します。そのポインターは Instrument の `setupArguments()` メソッドに渡されますが、このメソッドが Template Method パターンにおける可変部分の役割を果たします。このメソッドは、まず渡された引数（arguments 構造体）がその商品に適合するものかどうかチェックし、もしそうなら、その arguments 構造体に、自分のメンバー変数を代入していきます。最後に、arguments 構造体は、入力されたデータが有効範囲内のものかどうかを `validate()` メソッドを使って行います。

この段階で、Strategy Pattern の用意ができました。引数が準備できたので、Pricing Engine オブジェクトは、その商品の価格計算を、（その Pricing Engine 独自のアルゴリズムを実装した）`calculate()` メソッドを使って実行するよう要求されます。Pricing Engine は、arguments から受け取った入力データを使って計算を実行し、計算結果を results 構造体へ書き込みます。

Pricing Engine が以上の動作を完了した後、実行手順は Instrument オブジェクトに戻り、Template Method パターンが引き続き読み解かれます。すなわち `fetchResults()` メソッドが呼び出され、Pricing Engine に対し results 構造体を渡すよう要求し、そのオブジェクトが Downcast され、そのメンバー変数へのアクセスを可能にし、その内容を自らのメンバー変数にコピーします。Instrument ベースクラスは、すべての商品に共通の results 構造体をデフォールトで定義しており、各派生クラスで、さらに追加の計算結果を保有するよう拡張できます。

Aside: 純粋でない仮想関数

Listing 2.6を見て、`setupArguments()` メソッドが、純粋仮想関数として宣言されず、例外処理を行うよう実装されているのを、不思議に思われるかもしれません。(訳注：`setupArguments()` は実装されていませんというエラー処理しか行わないメソッドを作る意味があるのか、純粋仮想関数にして、派生クラスで実質な内容を実装すればいいのではないか、という疑問)　その理由は、開発者が、派生クラスにおいて `performCalculations()` を Override して実装するだけの場合は、意味のないメソッドをその派生クラスの為に実装するのを強制しないようにするためです。

3.2.1 例：シンプルなオプション

ここで、ひとつ例を示す必要があるでしょう。一言だけ注意をしておくと、QuantLib の中にプレインバニラーすなわち単純な株のコール・プット　オプションで、行使タイミングがヨーロピアンやアメリカンやバーミューダタイプーのオプションクラスは、深いクラス階層の奥底に存在します。ベースクラスとして Instrument クラスがあり、そこからまず Option クラスが派生し、さらに対象資産が一個のオプションを定義する OneAssetOption クラスが派生し、さらに一つ二つのクラスを経由して最終的に VanillaOption クラスに行きつきます。

この様に、多くの層を経由させた理由は当然あります。OneAssetOption クラスの定義は他のオプション、例えばアジアンオプションでも使えますし、Option クラスは、様々なタイプのバスケットオプションに使えます。残念ながら、そのせいで、単純なオプションの価格計算のプログラムコードが、多層な階層に分散されており、解りやすく説明する為の実例としては使いにくいものにな

っています。そこで、ここでの説明だけの為に、実装内容は QuantLib の Library と同じものですが、Instrument クラスから直接派生させた仮想の VanillaOption クラスを作り、中間にある派生クラスの実装内容もその中に取り込んでしまいました。

Listing 2.7にこの仮想 VanillaOption クラスの実装内容を示します。まず Instrument ベースクラスのインターフェースで実装が必要なメソッドを宣言し、次にこのクラスで追加された計算結果のデータ（名前にある通り、オプションの Greeks(感応度)）を読みだせるメソッド群を宣言しています。前の章で指摘した通り、それらのデータは `mutable` で宣言されており、論理的に `const` 宣言された `calculate()` メソッドによっても、値を変更出来ます。

Listing 2.7: Interface of the `VanillaOption` class.

```
class VanillaOption : public Instrument {
  public:
    // accessory classes
    class arguments;
    class results;
    class engine;
    // constructor
    VanillaOption(const boost::shared_ptr<Payoff>&,
                  const boost::shared_ptr<Exercise>&);
    // implementation of instrument method
    bool isExpired() const;
    void setupArguments(Arguments*) const;
    void fetchResults(const Results*) const;
    // accessors for option-specific results
    Real delta() const;
    Real gamma() const;
    Real theta() const;
    // ...more greeks
  protected:
    void setupExpired() const;
    // option data
    boost::shared_ptr<Payoff> payoff_;
    boost::shared_ptr<Exercise> exercise_;
    // specific results
    mutable Real delta_;
    mutable Real gamma_;
    mutable Real theta_;
    // ...more
};

class VanillaOption::arguments
    : public PricingEngine::arguments {
  public:
```

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

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

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

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

VanillaOption クラスは、自分自身に必要なメンバー変数やメソッドの他に、いくつかのアクセサリークラスを宣言しています。すなわち、このクラス特有の（内部クラスとなる）arguments と results 構造体、および基本となる PricingEngine クラスです。これらのクラスはすべて、Option クラスとの関係性を明確にするために、VanillaOption クラスの内部クラスとして定義されています。それらのインターフェースについても、上記 Listing の中で示されています。

これらのアクセサリークラスについて、2点コメントします。一点目は、この例を紹介した際に私が述べたことに反し、（クラス内クラスとして、このクラス特有の計算結果を保持するために定義されたはずの）`VanillaOption::results` クラスの中で、メンバー変数を全く宣言していません。これは、実装内容の詳細な仕組みを際立たせる為に、わざとそうしました。プログラム開発者は、いくつかの商品に共通するような計算結果を保持する構造体を定義したいと考えるかもせれません。そのような構造体は、派生クラスを作ることにより再利用が可能になるからです。Greeks 構造体の例がまさにそれです。（訳注：様々なオプションで共通のリスク感応度の値を、Greeks という名前の構造体で定義しておけば、オプション毎に results 構造体のメンバー変数を定義せず、Greeks 構造体を再利用すれば簡単で済む。）従って、このクラス（`VanillaOption::results`）は、Greeks 構造体と `Instrument::results` クラスから派生させて、最終的な構造が構築されています。その際に、かの悪名高い菱形継承を避ける

為、(Greeks::results も Instruments::results も)PricingEngine::results クラスからの仮想継承にしなければなりません。(継承のダイアモンドについては、例えば巻末(Stroustrup, 2013) 参照)(訳注:VanillaOption::results は、Instrument::results と Greeks::results の2つのベースクラスから派生しているが、この2つのベースクラスは、いずれも PricingEngine::results から派生している。この場合、仮想継承にしておかないと、派生クラスからベースクラスのメソッドを呼び出す時、2つのルートが発生して曖昧さが発生する。)

二点目は、この VanillaOption クラス特有の Pricing Engine クラスは、テンプレートクラスである GenericEngine から直接継承し、適切な arguments クラスと results クラスを使ってインスタンスを生成するだけで十分であったという事です。以下に説明するとおり、派生クラスは、ただ単に calculate() メソッドを実装するだけで十分です。

さあ、いよいよ VanillaOption クラスの実装内容(Listing 2.8)の説明に移りたいと思います。

Listing 2.8: Implementation of the VanillaOption class.

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

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

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

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

void VanillaOption::fetchResults(
                 const PricingEngine::results* r) const {
    Instrument::fetchResults(r);
    const VanillaOption::results* results =
        dynamic_cast<const VanillaOption::results*>(r);
```

```
        QL_ENSURE(results != 0, "wrong result type");
        delta_ = results->delta;
        ... // other Greeks
    }

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

コンストラクターは、具体的なオプションの条件を決める複数のオブジェクトを引数として取ります。これらについては後々の章または Appendix A で説明します。今の段階で簡単に説明しておくと、(引数のひとつ)payoff オブジェクトはストライク(行使価格)とオプションタイプ(コールかプットか)の情報を持ち、exercise オブジェクトは行使日と行使タイプ(ヨーロピアン、アメリカン、バーミューダン)の情報を持ちます。引数で渡されたこれらオブジェクトのインスタンスは、メンバー変数に保持されます。ここでは、市場データの情報は渡されていない点に注意して下さい。市場データは別の所で渡されています。

オプション期日到来に関するメソッドは極めて単純です。`isExpired()` メソッドは、最終オプション行使日が過ぎていないかどうかチェックし、`setupExpired()` メソッドは、ベースクラスで実装されたメソッドを呼び出すと同時に、いくつかのメンバー変数(Greeks)を0にセットします。

`setupArguments()` と `fetchResults()` メソッドの実装内容はもう少し複雑です。`setupArguments()` メソッドは、まず引数として受け取った汎用 arguments(ここでは `PricingEngine::arguments*`)のポインターを VanillaOption::arguments 型へダウンキャスト(型変換)します。もし、型が合っていない場合は、そこで例外処理に飛び、型が合っている場合は動作が継続します。次に。VanillaOption オブジェクトのメンバー変数は、逐語的に(型変換後の)arguments オブジェクトのメンバー変数にコピーされていきます。しかし、別々の Pricing Engine で、同じような計算(例えば、日付を時間に換算する計算)が必要な場合があるかも知れません。そういった場合は、`setupArguments()` のメソッドの中で一回実装するだけで済みます。

`fetchResults()` メソッドは、`setupArguments()` メソッドと対を成すものです。このメソッドも、引数として受け取った results オブジェクトのポインターをダウンキャスト(型変換)する事から始まります。型の適合性の確認を行った後、results オブジェクトの内容を自分のメンバー変数にコピーしていきます。

シンプルではありますが、上記のような実装で、実際に金融商品を稼働することができます。稼働するとは、Pricing Engine を備えれば、必要な価格計算ができるという事です。そのような Pricing Engine の例を、Black-Scholes-Merton によるヨーロピアンオプションの解析解モデルで実装したプログラムコードを Listing 2.9に示します。

Listing 2.9: Sketch of an engine for the `VanillaOption` class.

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

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

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

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

コンストラクターは、Black Scholes モデルの確率過程を記述するオブジェクトを引数として取ります。このオブジェクトは、市場データ（対象資産とその現在価値、リスクフリー金利、配当利回り、ボラティリティ）の情報を保持しており、それらをメンバー変数にコピーします。ここでも又、実際の計算過程は、他のクラス（BlackCalculator クラス）のインターフェースの奥に隠れてしまいましたが、プログラムコードを見れば、計算に必要なデータのやり取りの様子はわかると思います。

`calculate()` メソッドは、まずいくつかの事前データチェックからスタートします。これには少し驚かれるかもしれません。なぜなら、これらのデータは `calculate()` がスタートする時点ですでにチェック済のはずだからです。しかし、どんな Pricing Engine でも、計算をスタートする前に、さら

なるデータチェックをする場面があるかもしれません。上記のプログラムでは、オプションがヨーロピアンか否か、payoff が単純な Call または Put なのか、というチェックを行い、後者の手続きはさらに、必要とするクラスに型変換するプロセスまでも含みます。(`boost::dynamic_pointer_cast` は、shared pointer の為の `dynamic_cast` と同じです。)

このメソッドのプログラムコードの真中あたりに、Pricing Engine が引数から必要な情報を取り出している部分があります。ここでは、対象資産の現在価格やその他の価格計算に必要な情報(対象資産のオプション期日での先渡し価格、満期時における Discount Factor)などです。(QuantLib のソースコードにすべて記述されているので、確認して下さい)

最後に、計算アルゴリズムが実行され、計算結果が results 構造体の該当箇所にそれぞれコピーされます。これで、`calculate()` メソッドと事例の紹介を終わります。

4. Term Structures

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

4.1 The `TermStructure` Class

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

4.1.1 インターフェースと要件

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

4.1.2 実装

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

4.2 金利の期間構造

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

4.2.1 インターフェースと実装

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

4.2.2 ディスカウントカーブ、フォワード金利カーブ、ゼロ金利カーブ

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

4.2.3 例：Interpolation されたカーブの Bootstrapping

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

4.2.4 例：イールドカーブに z ースプレッドを追加：

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

4.3 その他の期間構造クラス

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

4.3.1 デフォールト確率の期間構造

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

4.3.2 インフレ率の期間構造

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

4.3.3 ボラティリティの期間構造

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

4.3.4 株式ボラティリティの期間構造

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

4.3.5 金利ボラティリティの期間構造

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

5. Cash Flows and Coupons

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

5.1 The `CashFlow` Class

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

5.2 Interest-Rate Coupons

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

5.2.1 固定金利クーポン

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

5.2.2 変動金利クーポン

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

5.2.3 例：LIBOR クーポン

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

5.2.4 例：Cap・Floor 付きのクーポン

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

5.2.5 キャッシュフロー配列の生成

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

5.2.6 他の種類の **Coupon** と今後の開発

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

5.3 キャッシュフローの分析

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

5.3.1 例：固定金利債券

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

6. Parameterized Models and Calibration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

6.1 CalibrationHelper クラス

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

6.1.1 例： Heston モデル

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

6.2 モデルのパラメータ

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

6.3 CalibratedModel クラス

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

6.3.1 例：Heston モデル（続き）

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

7. The Monte Carlo Framework

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

7.1 Path の生成

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

7.1.1 乱数の生成

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

7.1.2 確率過程

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

7.1.3 Path の生成装置

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

7.2 Path 上での価格計算

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

7.3 すべての部品を使って組み立てる

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

7.3.1 Monte Carlo traits　クラス

7.3.2 モンテカルロモデル

7.3.3 モンテカルロシミュレーション

7.3.4 例: バスケットオプション

8. Tree を使った価格モデルのフレームワーク

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

8.1 格子クラスおよび離散化資産クラス

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

8.1.1 例：Discretized Bond （離散モデルで表現された債券）

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

8.1.2 例：DiscretizedOption クラス（離散モデルで表現されたオプションクラス）

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

8.2 Tree および　Tree-Based Lattice

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

8.2.1 Tree クラスのテンプレート

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

8.2.2 ２項 Tree および３項 Tree クラス

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

8.2.3 TreeLattice クラステンプレート

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

8.3 Tree をベースにした価格エンジン

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

8.3.1 例： コールオプション付き固定金利債

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

9. 有限差分法のフレームワーク

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

9.1 古いフレームワーク

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

9.1.1 微分演算子

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

9.1.2 時間軸方向の差分スキーム

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

9.1.3 境界条件

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

9.1.4 Step Condition: 時間ステップ遷移時の条件

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

9.1.5 有限差分モデルクラス

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

9.1.6 例：アメリカンオプション

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

9.1.7 時間に依存する差分演算子

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

9.2 新しいフレームワーク

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

9.2.1 メッシャー（離散化した確率変数の格子）

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

9.2.2 差分演算子

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

9.2.3 例：Black-Scholes モデル用の差分演算子

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

9.2.4 初期条件、境界条件　および　時間ステップ条件

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

9.2.5 時間軸方向の差分スキームとソルバー

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

10. おわりに

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/implementingquantlib-jp.

11. Appendix A: 周辺の話題

QuantLib ライブラリーを使うにあたって、これまでの Chapter で、基本的な事項について説明して来ましたが、読みやすさを重視して、技術的な詳細について延々と述べる事を避けてきました。Douglas Adams が Hitchhiker 三部作（銀河ヒッチハイクガイド）の、4番目の本の中で、次のように指摘しています。（注：3部作における4番目の本は変ですが、間違いではありません。そもそも5冊の本からなるのに3部作と呼ばれています。）

「必要以上に詳細な説明は問題です。それは行動を前に進めません。詳細な説明は本を分厚くしますが、あなたが行きたい所にはたどり着かせないでしょう。」

この Appendix は、そういった詳細な事項について、簡単な参考説明を提供しています。しかし、それらを網羅的かつ体系的に説明しようとするものではありません。もし、そのような参考文献が必要なら、QuantLib ライブラリーの中にあるReference Manual を参照下さい。

11.1 基本的なデータ型

QuantLib ライブラリーのインターフェースは、（`int` や `float` や `double` のような）既に組み込まれているデータ型を使っていません。その代わりに、typedef を使って、`Time`、`Rate`、`Integer`、や `Size` といった QuantLib ライブラリー特有のデータ型を定義しています。これらのデータ型は、すべて基本的なデータ型に対応付けされています。（以前、型のレンジチェックの機能まで付いたデータ型を定義しようか議論しましたが、それはあきらめました。）さらに、すべての浮動小数点型は `Real` と定義され、それは `double` に対応しています。こうすれば、`Real` に対応するデータ型を定義し直すことで（訳注：例えば `double` を `long double` に定義し直して）、整合的にすべての `Real` 型変数の浮動小数点の精度を同時変更できます。

原理的には、これにより、浮動小数点の数値精度をユーザーが選択できます。しかし、それが理由で、test-suite の中で、`Real` 型を、`float` や `long double` で定義し直した場合に、変な計算結果が出るケースが発生してしまいました。typedef を使うメリットは、プログラムコードの意味がより明瞭になる事です。また、私のように、かつて物理学者で次元解析に馴染みがあれば、例えば、`exp(r)` とか `r+s*t` といった式が `Rate r, Spread s, Time t` という型宣言の後に来ると、直ぐに変だと気づく事ができます。（訳注：`r` が金利、`s` がスプレッド、`t` が時間を示しているのが、データ型からすぐ判れば、`exp(r)` は、`exp(r*t)` あるいは `exp(-r*t)` の間違いで、`r+s*t` は `(r+s)*t` の間違いだとすぐ気づく事ができます）

もちろん、これらのデータ型は、`double` と同義語であり、コンパイラーは全く同じものとして扱います。もし、これらのデータ型がより強い型としての機能を持つなら、もっとよかったかも知れません。例えば、あるメソッドが Price を渡された場合と Volatility を渡された場合に、（いずれも `double` であるにも拘わらず）異なった型の入力データとして取り扱い、関数のオーバーロードが出来るとか。

それを可能にする手段として、BOOST_STRONG_TYPEDEF を使う方法があり、これは Boost ライブラリーが提供している膨大な機能のひとつです。これは、例えば、次のようにして使います。

```
BOOST_STRONG_TYPEDEF(double, Time)
BOOST_STRONG_TYPEDEF(double, Rate)
```

こうすると、対象とするデータ型と同じように使える、新しいクラス型が創出できます。この方法だと、先ほどのような関数のオーバーロードが可能になります。一方で、この方法の欠点は、すべての型変換が明示的では無い事です。そうすると、古いバージョンとの互換性に問題が発生し、プログラミング上も面倒くさいことになります。

（注：例えば、単純な構文 `Time t = 2.0;` でもコンパイルエラーになる可能性があります。また、`Time` を引数として取る関数 `f()` について `f(1.5)` では無く `f(Time(1.5))` という風に書く必要があります。）

また、（BOOST_STRONG_TYPEDEF のような）マクロによって定義されたクラス型を使うと、すべての演算子をオーバーロードします。そうすると、Time（時間）に Rate（金利）を足すような事も出来てしまいます（そう、これもまた次元解析的には問題です）。もし、データ型の体系が、Rate と Spread の足し算や Rate と Time の掛け算は許しても、Rate と Time の足し算に対してはコンパイルエラーを出してくれると助かります。

このような事をランタイムでの計算時間に影響を与えず、かつ汎用的に行えるような枠組みは、最初に Barton and Nackman, 1995 によって示されました。このアイデアの発展形は Boost.Units ライブラリで実装されています。より簡単なバージョンは、私が物理学の世界で働いていた時代に書いたコードがあります（ここでは、説明しませんが、なかなかいい出来です）。しかし、そこまでの機能は、QuantLib の中では必要ないでしょう。QuantLib では、長さ、体積、質量、時間などの物理量をすべて取り扱う必要は無いからです。

将来取り入れる可能性がある理想的な妥協点としては、（Boost の strong typedef のような）Wrapper クラスを実装する方法があります。そのクラスは明示的に、どの演算子がどの型に許されているか定義します。いつもの事ですが、このアイデアを示したのは我々が最初ではありません。次のバージョンの C++ に加えるべき機能として、opaque typedef という名前で、(Brown, 2013) により提案が為されています。それが実現すれば、この種の型を定義するのは、より簡単になるでしょう。

最後の注意点として、これらの型の中で、（型名から変数の内容が）一意に決まらない型タイプがあります。価格の volatility と金利の volatility は、異なる次元の値と考える必要があります。従って、両者は異なる型として取り扱うべきです。端的に言えば、`Volatility` 型はテンプレート型にすべきでしょう。

11.2 Date Calculations: 日数計算方法

日数計算の機能は、クオンツファイナンスにおける基本的な道具のひとつです。当然ながら、QuantLib ライブラリーも、いくつかの日数計算の機能を提供しています。以下のセクションで、これらについて簡単に解説します。

11.2.1 日数と期間を扱うクラス

Date クラスのインスタンスは、個別の日付（例えば2014年11月15日）を表現するオブジェクトです。このクラスは、いくつかのメソッドを用意しており、該当する日付に関する情報（たとえば営業日に該当するかどうか、ある月や年の何日目になるのか、等）を取りだす事ができます。その他、個別の日付で許容されている最小と最大の日付（今の所、それぞれ1901年1月1日と2099年12月31日に設定されています）、その年が閏年かどうか、その日を Excel で使っている日付の Serial ナンバーに換算した値、その日が月末に該当するかどうか、といった情報も取りだす事ができます。提供されているメソッドとそのインターフェースの全リストは、QuantLib ライブラリーの Reference Manual に掲載されています。但し、「時間」の情報（その日の何時何分か）は提供されていません。（今、それを加えるか検討中です。）

Date クラスでは、C++ の機能を活用して、算術演算子をオーバーロード関数で定義しており、日付に関する代数計算を自然に行う事ができます。例えば、`++d` という表現は、特定の日 `d` を一日前に進める操作になり、`d+2` は `d` を 2日分先に進める操作を表し、`d2-d1` はふたつの日付 `d1` と `d2` の間の日数を表し、`d-3*Weeks` は d の3週間前の日付を表します（`Weeks` は `TimeUnit` の enumeration で提供されているカレンダー単位のひとつで、他の enum メンバーには `Days`、`Months`、`Years` があります）。また `d1<d2` という比較演算子は、`d1` が `d2` の前にあるなら `true` を返します。Date クラスが提供しているこういったオーバーロードされた演算子の機能は、カレンダー日付を前提にしており、休日や営業日調整の慣行は考慮されていません。

`Period` クラスは、2日や3週間や5年といった期間の長さをオブジェクトモデル化したもので、`TimeUnit`（日、週、月、年といった期間の単位）と、その長さを表す整数を組合せて、データを保存しています。このクラスが提供している、オーバーロードされた算術演算子と比較演算子は限られています。理由は、カレンダーは完全な算術計算に馴染まない傾向がある為、2つの `Period` インスタンスについてどちらが長いか短いか、簡単に比較できないからです。例えば、11か月は1年より短いのは明らかですが、60日が2か月より短いか長いかは、どの2か月を取るかで異なります。仮に、比較がうまく出来ない場合は、例外処理に飛ぶようになっています。

比較内容が明らかな場合であっても、いくつかの驚くような事を克服する必要があります。例えば次のような比較ですが

```
Period(7,Days) == Period(1,Weeks)
```

は、`true` を返します。当然、正しい結果と思いますよね？　覚えておいて下さい。

11.2.2 カレンダークラス

`Calendar` クラスは、休日と営業日の情報を取扱います。このクラスには、数多くの派生クラスが存在しており、それらは具体的な市場毎の休日の情報を定義しています。ベースクラスは、いくつかのメソッドを提供しており、特定の日が休日か営業日かを判定するシンプルなメソッドや、仮にその日が休日だった場合に、最も近い営業日を特定する少し複雑なメソッド（最も近いという操作は、`BusinessDayConvention` の enumeration に従って決定されます）、さらに、与えられた日数あるいは営業日数をもとに、カレンダーをその期間だけ前に進めるメソッドなどです。

Calendar クラスの記述の仕方によって、その動作がどのように異なるのかを見てみるのは興味深い事です。ひとつの方法は、Calendar インスタンスに、その特定の市場の休日情報をすべてリストで持たせる事です。しかし、この方法だと休日の変更があった場合などメンテナンスの問題が発生します。

従って、我々は、休日決定のルール（例えば、11月の第4木曜日とか、各年の12月25日とかいう決まり）をプログラムコード化する方法を取るべきと考えました。そうすると、polymorphism（多相性）を持たせた Template Method パターンを使うのが有力な方法です。この方法では、派生クラスで、ベースクラスの `isBusinessDay()` メソッドをオーバーライドして、様々な国のカレンダーに対応します。これでも OK ですが、欠点として Calendar インスタンスを、`shared_ptrs` を使って、他のオブジェクト等に渡したり、データを保持したりする必要があります。このクラスは概念的にもシンプルで、しかも頻繁に使われるオブジェクトなのでユーザーが簡単にインスタンス化し、利用できるようにした方が良く、そうすると (`shared_ptrs` による) メモリーの dynamic allocation の仕組みは余計な冗長さになるかも知れません。

最終的に我々が取った方法は、下記 Listing A.1に示すような形です。これは pimple idiom (訳注：Pointer to Implementation idiom の略) のバリエーションで、Strategy パターンや Bridge パターンの名残でもあります。最近の若い人たちは、type erasure（型消去のテクニック）とも呼んでいるようです。

Listing A.1：`Calendar` クラスの概要

```
class Calendar {
  protected:
    class Impl {
      public:
        virtual ~Impl() {}
        virtual bool isBusinessDay(const Date&) const = 0;
    };
    boost::shared_ptr<Impl> impl_;
  public:
    bool isBusinessDay(const Date& d) const {
        return impl_->isBusinessDay(d);
    }
    bool isHoliday(const Date& d) const {
        return !isBusinessDay(d);
    }
    Date adjust(const Date& d,
                BusinessDayConvention c = Following) const {
        // uses isBusinessDay() plus some logic
    }
    Date advance(const Date& d,
                 const Period& period,
                 BusinessDayConvention c = Following,
                 bool endOfMonth = false) const {
```

```
        // uses isBusinessDay() and possibly adjust()
    }
    // more methods
};
```

手短に説明すると、Calendar クラスは polymorphic な内部クラス Impl を宣言し、営業日決定(or 休日決定)ルールの実装をその派生クラスに委任し、自らはその pointer を保持するものです。isBusinessDay() メソッドは、それ自体は仮想関数ではありませんが、(関数内で Impl のポインターを使って)動作を、Impl 派生クラスのメソッドに委託しています。その他のメソッドも、Template Method パターンをある程度取り入れ、仮想関数としてでは無く、isBusinessDay() メソッドを直接あるいは間接的に使って実装されています。
(注:同じテクニックは次のセクションの DayCounter クラスや、Chapter VI で説明した Parameter クラスでも使われています。)

派生 Calendar クラスは、Calendar::Impl の派生クラスを内部クラスとして定義し、具体的なカレンダーの動作を提供します。そのコンストラクターは、Impl インスタンスへの shared pointer を生成し、それをベースクラスの impl_ に保持します。その結果、生成された Calendar インスタンスは、それを使いたいどんなオブジェクトであっても、安全にコピーできます。このインスタンスを分割しても、polymorphic な Impl クラスへのポインターのおかげで、正しい動作を維持する事ができます。最後に、同じ派生 Calendar インスタンスは、同じ Impl インスタンスをシェアできる点を注意しておきます。これは Flyweight パターンの実装とみる事ができ、ひとつの単純なクラスの為に、合計で2.5種類のデザインパターンを使ったことになります。

Calendar クラスの実装内容の説明については十分したので、その動作の説明に移ります。前のセクションで説明した、"驚くべき事"についてです。Period(1,Weeks) は Period(7,Days) と同じ(== 演算子が true を返す)と述べた所を覚えていますか?実は、Calendar クラスの advance() メソッドだけは、7 Days は7 business Days(7営業日)として計算しています。従って、仮に2つの期間 p1 と p2 が同じ(すなわち p1==p2 が true を返す)だったとしても、calendar.advance(p1) の結果は calendar.advance(p2) と違ってくる可能性があります。とんでも無い事をやってしまいました。

この問題に対する良い解決策を持っている訳ではありません。過去のバージョンとの互換性を考えれば、今の advance() メソッドでの Days の使い方はそのままにしなければなりません。そうすると、(7カレンダー日だけ日を前に進めたい場合でも)calendar.advance(7, Days) として、それを7カレンダー日前に進めたと解釈する事はできません。ひとつの逃げ道は、今の状況はそのまま残して、enumeration の中に BusinessDays と CalendarDays を追加する方法です。(今後開発されるプログラムコードで、これを使い分ければ、)意味の曖昧さが取り除かれ、次第に Days を使われなくなるでしょう。あるいは、(advance() メソッドも含めて QuantLib 全体で)7 days は1 week とは違うという事に統一してしまう方法です。しかし私自身はあまり乗り気ではありません。

もし、過去のバージョンとの互換性をあきらめるなら(待ち遠しい QuantLib 2.0のバージョンで)、もっと別の解決策があります。ひとつは、Days を常にカレンダー日数として使い、BusinessDays は営業日数を示すものとして(TimeUnit の)enumeration に加える方法です。別の方法は、(私自身は、考えれば考えるほどこちらの方法がいいと思いますが)、Days を常にカレンダー日数として使い、Calendar クラスに advanceBusinessDays() メソッドを追加するか、advance() メソッドをオーバ

ーロードして `advance(n, BusinessDays)` を加える方法です（ここで `BusinessDays` は別のクラスのインスタンスとする）。しかし、これは例えば3 business days は、もはや期間でなくなる（年数換算の計算が出来ない）ことになります。

既に申し上げた通り、明解な解決策はありません。もし読者の方が別の解決法をお持ちなら、お聞かせ下さい。

11.2.3 Day Count Conventions: 日数計算方法

`DayCounter` クラスは、2つの日付の間の期間を計算するツールを提供しており、その期間は、“日数”あるいは“小数点付きの年数”で表されます。`Actual360` や `Thirty360` といった派生クラスが存在しており、これら派生クラスは前のセクションで説明した Calendar クラスと同じ様な polymorphic（多相的）な動作を実装しています。
（訳注：様々な日数計算方法については、Wiki 参照）

そのインターフェースは、残念ながら少し粗い作りです。例えば、`yearFraction()` メソッドは、単に2つの日付を取るのでは無く、次のようになっています。

```
Time yearFraction(const Date&,
                  const Date&,
                  const Date& refPeriodStart = Date(),
                  const Date& refPeriodEnd = Date()) const;
```

2つの任意引数（refPeriodStart と refPeriodEnd）は、ある特別の日数計算方法（ISMA Actual/Actual がそれです）の為だけに必要です。この日数計算方法では、2つの日付の他に2つの reference 日付けを指定する必要があります。（訳注：ISMA Actual/Actual は、主に米国債の経過利息計算に使われているが、分母にあたる実日数は、クーポン期間の実日数で、年2回払いであれば、182日であったり、183日であったりする）

ところが、共通のインターフェースを持たせる為に、（ISMA Actual/Actual 以外の）派生クラスでも、`yearFraction()` メソッドの宣言に、この2つの追加の引数を加えなければなりませんでした（ほとんどのクラスで、問題なくそれを無視していますが）。この DayCounter が起こした問題は、これだけに留まりません。それらを次のセクションで見てみます。

11.2.4 Schedules: クーポンスケジュール

次の Listing A.2に示す `Schedule` クラスは、クーポン日付のスケジュールを生成する為に使われます。

Listing A.2:Schedule クラスのインターフェース

```
class Schedule {
  public:
    Schedule(const Date& effectiveDate,
             const Date& terminationDate,
             const Period& tenor,
             const Calendar& calendar,
             BusinessDayConvention convention,
             BusinessDayConvention terminationDateConvention,
             DateGeneration::Rule rule,
             bool endOfMonth,
             const Date& firstDate = Date(),
             const Date& nextToLastDate = Date());
    Schedule(const std::vector<Date>&,
             const Calendar& calendar = NullCalendar(),
             BusinessDayConvention convention = Unadjusted);

    Size size() const;
    bool empty() const;
    const Date& operator[](Size i) const;
    const Date& at(Size i) const;
    const_iterator begin() const;
    const_iterator end() const;

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

様々な市場慣行や ISDA の決まり事を取り込む為、このクラスは相当数のパラメータを取る必要があります。それがいかに多いかは、コンストラクターの引数のリストを見れば一目瞭然です。(これらの引数の説明をしませんがお許し下さい。意味については皆さんお判りでしょう。)この数はおそらく多すぎるので、QuantLib では Named Parameter Idiom(既に Chapter IV キャッシュフロー配列の生成の所で説明しました)を使い、より使いやすい Factory クラスを提供しています。それを使えば、次のようなコードの記述で、Schedule クラスのインスタンスが生成できます。

```
Schedule s = MakeSchedule().from(startDate).to(endDate)
             .withFrequency(Semiannual)
             .withCalendar(TARGET())
             .withNextToLastDate(stubDate)
             .backwards();
```

このクラスは他にも、メンバー変数のデータを読み取るインスペクター関数や、日付の配列に対するインターフェースとして `size()`、`operator[]`、`begin()`、`end()` といったメソッドを提供しています。

`Schedule` クラスは、もう一つ別のコンストラクターがあり、引数として事前に設定された日付の配列を取ります。但し、これについては未完成です。このコンストラクターを使って生成されたインスタンスは、インスペクター関数に対応するデータが揃っておらず、今の所、そういった場合は例外処理に飛びます。その例として、`tenor()` や `isRegular()` といったメソッドがあり、それについて少し説明する必要があります。

まず `isRegular(i)` メソッドですが、このメソッドが指し示すのは i 番目のクーポン日付では無く、i 番目のクーポン期間、すなわち i と i+1番目のクーポン日の間の期間を指します（訳注：その期間が regular なら true、そうでなければ false を返す）。そう言いましたが、では”regular”はどういう意味なのでしょうか？クーポンスケジュールが、“クーポン期間”をベースに生成された場合、大半のクーポン期間は引数で指定された期間と同じになります（それが regular の意味です）。しかし、最初と最後のクーポン期間は、最初のクーポン日あるいは最後からひとつ手前のクーポン日を明示的に指定した場合、指定されたクーポン期間より長かったり短かったりします。例えば、First Short Coupon 日を特に設定したい場合などは、その日を明示的に指定します。

もし、Schedule が、事前に設定されたクーポン日の配列でインスタンス化された場合、上記のような指定された regular なクーポン期間情報を持たない為、`isRegular(i)` メソッドの問いかけに答えられません。より問題なのは、それにより、この Schedule インスタンスを使って、実際の債券クーポンのスケジュールを生成する事が出来なくなる事です。もし、この Schedule インスタンスを、固定クーポン債を生成するコンストラクターに渡した場合、例外処理に飛んでしまいます。

では、なぜ債券のコンストラクターは、クーポンスケジュールを生成するのに、この `isRegular(i)` の情報が必要なのでしょうか？理由は、もしその債券のクーポンの日数計算方法が ISMA actual/actual だった場合、reference 期間の情報が必要になりますが、その reference 期間の情報を計算する為には、クーポン期間の情報も必要になるからです。

幸いなことに、この問題を解決するのは難しくありません。ひとつは、日数計算方法をチェックし、必要な場合にだけ reference 期間を計算する事です。この場合でも、日数計算方法が ISMA actual/actual の場合は例外処理に飛びますが、その他の日数計算方法ならすべてうまく行きます。別の方法は、（クーポン日配列を渡されて生成された）Schedule インスタンスに、クーポン期間と regularity の情報を何とかして導出して加える事です。そうすれば、対応するメソッドはすべてうまくいきます。しかし、この方法に意味があるかどうか、確信がありません。

11.3 金融に関連する概念を対象とするクラス

QuantLib ライブラリーが対象としている領域からすれば、金融に関する概念をオブジェクトモデル化したクラスがいくつかあるのは当然です。その中のいくつかについて、このセクションで説明します。

11.3.1 Market Quotes: 市場データ

市場でクォートされている金融商品の価格をオブジェクトモデル化するには、少なくとも2通りの方法があります。ひとつは、市場価格を、Time-stamp（価格を取得した時間）と対応させた連続した価格の配列とし、配列の最後に直近の値に対応するようにしたものです。もうひとつは、現時点の価格をモデル化し、その値が価格変動に合わせ動的に動くようにオブジェクトモデル化したものです。

いずれの方法も有用であり、QuantLib ライブラリーでは両方のオブジェクトモデルを実装しています。最初のモデルは `TimeSeries` クラスに対応していますが、ここでは詳細な説明を控えます。このクラスは基本的に、日付と価格を対応づけて、そこから指定された日付に対応する価格を取りだすメソッドや、価格の配列を操作する Iterator が備わっています。しかし、このクラスは QuantLib ライブラリーの中で、他のクラスで使われた形跡はありませんでした。

もうひとつのモデルは、Quote クラスです。その概要を下記 Listing A-3 に示します。

Listing A.3 : `Quote` クラスのインターフェース

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

ご覧の通り、インターフェースはスリムです。このクラスは `Observable` クラスから派生しており、値（市場価格）が変更になった場合は、この市場価格に依存している（register されている）他のオブジェクト (Observer) すべてに変更通知を送ります（訳注：ベースクラスの Observable で定義されている `notifyObserver()` が呼び出される）。このクラスは `isValid()` メソッドを宣言しており、保持する市場価格が有効な値かどうかを返します（価格が有効期限を過ぎているといったチェックをするのではありません）。また `value()` メソッドは、現時点の市場価格を返します。

この2つのメソッドで、Quote クラスが必要とされる機能が十分提供されています。市場価格に依存しているすべてのオブジェクトは（例えば Chapter II で説明した Bootstrap Helper クラスなどは）、Quote オブジェクトの Handle（訳注：メモリー管理を自動で行うポインターへのポインター）を保持し、かつ自らを Observer として Quote クラスに register（登録）します。その仕組みにより、Observer オブジェクトは、いつでも最新の市場データにアクセスできます。

QuantLib ライブラリーでは、Quote の派生クラスを数多く定義しており、ベースクラスのインターフェースを、そこで実装しています。その内のいくつかは、市場価格として他の Quote オブジェクトから取得した価格を、加工してから返すものもあります。例えば、`ImpliedStdDevQuote` クラスは、特定のオプション価格から Implied Volatility を計算し、それを返すようになっています。他には、他の市場データオブジェクトに付随させて使うものがあります。`ForwardValueQuote` オブジェクトは、金利の TermStructure オブジェクトを使って、先日付の金利インデックスの Fixing Rate を計算して返します。また、`LastFixingQuote` オブジェクトは、Fixing Rate の時系列配列から最新の値を返します（訳注：このケースでは時々刻々変化する動的な市場価格というより、一日一回更新される Fixing Rate の情報から直近の情報を取りだしている）。

現時点では、外部データを情報源とする Quote オブジェクトとして実装されているのはひとつしかありません。それが `SimpleQuote` クラスであり、その実装内容を下記 Listing A.4 に示します。

Listing A.4 : `SimpleQuote` クラスの実装内容

```
class SimpleQuote : public Quote {
  public:
    SimpleQuote(Real value = Null<Real>())
    : value_(value) {}

    Real value() const {
        QL_REQUIRE(isValid(), "invalid SimpleQuote");
        return value_;
    }

    bool isValid() const {
        return value_!=Null<Real>();
    }

    Real setValue(Real value) {
        Real diff = value-value_;
        if (diff != 0.0) {
            value_ = value;
            notifyObservers();
        }
        return diff;
    }

  private:
    Real value_;
};
```

このクラスは、具体的な市場データからのデータフィードを実装していないので、極めてシンプルです（訳注：実際に外部データからのデータフィードを使う場合、データ供給元が提供する API を使うことになります。その際、様々なチェックや、エラー処理の機能を備える必要があります）。

ここでは単に、新しい市場価格を、適切なメソッドを呼び出してマニュアルで更新するようになっています。直近の市場価格は（もしそれが無ければ `Null<Real>()` が)、メンバー変数 `value_` に格納されます。ベースクラスのインターフェースである `value()` メソッドが実装されており、格納された値を返すようになっています。また `isValid()` メソッドは、その値が Null 値か否かをチェックします。新しい市場価格を取りこむメソッドは `setValue()` で、その引数で新しい市場価格が供給されると、それが直近の価格と異なっている場合は、（新しい価格を value_ にコピーした上で）Observer オブジェクトに通知し、直近価格との価格差を返します。（注：価格差を返すようにしたのは、C や C++ の設計慣行からすれば（通常は更新前の値を返す)、おかしいかもしれません。）

最後にいくつかの注意点を述べて、このセクションを終わりにしたいと思います。

まずひとつ目は、`Quote` クラスが対象とする市場価格のデータ型は `Real`（訳注：実体は `double`）のみです。これまでの所、それで問題になった事はありませんが、もし他の型も使えるようにする為に `Quote` クラスを Template クラスとして定義し直そうとしても、もはや手遅れです。従って、この点についての変更が行われる事は無いでしょう。

ふたつ目は、当初の目論見は、この `Quote` クラスを、実際の市場価格データフィードとのアダプターとして使い、（データ供給元が異なる場合に）異なる API（Application Program Interface）の呼び出しに応じた実装をして、全体として統一的に市場価格をオブジェクト化しようというものでした。しかし、これまでの所、（QuantLib ライブラリーのユーザーの中で）誰もそのような実装を行っていないようです。それに一番近い使い方は、Excel に取り込んだ市場価格のデータフィードを使って `SimpleQuote` インスタンスの値を設定するくらいです。

最後の注意点は少し長いですが、`SimpleQuote` クラスのインターフェースは将来、より先進的な使い方をする為に修正される可能性があります。複数の市場価格を一纏めにして、新しい値をセットする場合（例えば、イールドカーブを Bootstrapping する為に使われる金利の Quote の配列など)、それぞれの金利 Quote が更新される都度ではなく、その Quote すべてが更新されてから、`notifyObserver()` を一括して発信する方が効率的です。なぜなら、Observable から Observer へ価格更新の通知と変更のプロセスは、結構時間がかかる為です。この変更は（`setValue()` メソッドに追加で `silent` というパラメータを渡し、それが `true` の場合は、`notifyObserver()` を呼び出さないようにする変更は）既に Library の開発者によって実装済で、いずれ QuantLib ライブラリーに加えられるでしょう。

11.3.2 金利

`InterestRate` クラス（下記 Listing A-5参照）は一般的な金利計算のロジックをカプセル化したものです。このオブジェクトのインスタンスは、4つの引数、すなわち ① 利率（Rate)、② 日数計算方法（Day Counter)、③ 複利の方法（Compounding method)、④ 複利の回数（Compounding Frequency)、を使って生成されます（但し、金利の複利の回数にかかわらず、利率（Rate）の表示は、年率換算後の値を使います)。これにより、使用される「金利」の、具体的内容が定まります。例えば、5%、Actual / 365、Continuously Compounded（連続複利）とか、2.5%、Actual /360、Semiannually compounded（年2回複利)、とかです。但し、複利の回数については必ずしも常に必要とされる情報ではありません。これについては後で説明します。

Listing A.5：`InterestRate` クラスの概要

```
enum Compounding { Simple,              // 1+rT
                   Compounded,          // (1+r)^T
                   Continuous,          // e^{rT}
                   SimpleThenCompounded
};

class InterestRate {
  public:
    InterestRate(Rate r,
                 const DayCounter&,
                 Compounding,
                 Frequency);
    // inspectors
    Rate rate() const;
    const DayCounter& dayCounter();
    Compounding compounding() const;
    Frequency frequency() const;
    // automatic conversion
    operator Rate() const;
    // implied discount factor and compounding after a given time
    // (or between two given dates)
    DiscountFactor discountFactor(Time t) const;
    DiscountFactor discountFactor(const Date& d1,
                                  const Date& d2) const;
    Real compoundFactor(Time t) const;
    Real compoundFactor(const Date& d1,
                        const Date& d2) const;
    // other calculations
    static InterestRate impliedRate(Real compound,
                                    const DayCounter&,
                                    Compounding,
                                    Frequency,
                                    Time t);
    ... // same with dates
    InterestRate equivalentRate(Compounding,
                                Frequency,
                                Time t) const;
    ... // same with dates
};
```

このクラスは、いくつかの自明なインスペクターの他に、数種類のメソッドを提供しています。ま

ず `Rate()` オペレーターですが、変数の型を（InterestRate 型から）`Rate` 型すなわち `double` に変換します。後で考えてみると、これは少しリスクのあるやり方でした。型変換された後の値は、「日数計算方法」や「複利の回数」など、金利に関する重要な情報を失ってしまうからです。そうすると、例えば連続複利利回りの情報が欲しかったにも拘らず、単純複利の情報がそのまま使われてしまうような事が起こりうるかもしれません。この型変換のオペレーターは（QuantLib の開発段階で）`InterestRate` クラスを導入した時、backward compatibility（古いバージョンのプログラムを使用した際に問題を起こさせない）の為に加えられたものでした。将来、どこかでこのオペレーターを取り除こうと考えていますが、ユーザーにとって、その方がより安全と思われる時にやりたいと考えています。

（注：安全性については、中心となっているソースコードの開発者にとって、意見が分かれるものです。ある者は、ベビーシッターのように、何なから何まで面倒をみないといけないと考えているし、ある者は、少しの遺産を与えて子を世に送り出す親のように、ユーザーがある程度自分で面倒を見るべきと考えています。）

その他のメソッドはすべて、（金利に関する）基本的な計算を提供しています。まず `compoundFactor()` メソッドですが、与えられた期間 t（あるいは2つの日付 d_1 と `d_2` の間の期間として与えられる）において、所与の金利と複利の回数に従って、元本1に対する倍率を返します。`discountFactor()` メソッドは、所与の期間、金利、複利の回数に従って、割引率（compoundFactor の逆数）を返します。`impliedRate()` メソッドは、所与の Compound Factor、日数計算方法、複利の回数、期間、から年率ベースの利回りを返します。`equivalentRate()` メソッドは、別の複利回数で計算した場合の利回りを返します。

`InterestRate` クラスのコンストラクターと同様に、上記のメソッドのいくつかは複利の回数（Frequency）を引数として取ります。しかし、場合によってはこの情報は不要です。そういう時のために、`Frequency` を定義している enum には、`NoFrequency` の項目も含めています。

この方法は Bug の原因になりやすいかもしれません。複利回数の情報（Frequency）は、複利の方法の情報（Compounding）と一体となって持たせて、それと関係のない方法（例えば、単利や連続複利）では、そういった情報を一切持たせないのが理想かもしれません。もし C++ が以下のような Syntax（構文）を持つなら、うまくいくのですが、残念ながらそうなっていません。

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

この Syntax は、関数言語における代数データ型や、Scala における case classes のようなものですが C++ では選択肢にありません。（注：両方とも、C++ の switch 構文をより強力にした、オブジェクトにおける型を適合させる機能を持つ。興味のある方はそちらの方を一読下さい）

C++ で同じような機能を持たせるためには、Strategy パターンを使い、`Compounding` クラスの階層を作れば出来なくはありません。しかし、それはやりすぎのような気がしますので、enum を使った構文のままにし、若干の問題を甘受することにしました。

11.3.3 Index: インデックスクラス

Index クラスは、Instrument や TermStructure といったクラスと同様、非常に広い概念をカバーします。このクラスは、例えば、金利インデックス、インフレーションインデックス、株価インデックス等々、皆さんご存知の概念をカバーしています。

言わずもがなですが、モデル化された概念は非常に多くのものをカバーしている為、共通となるインターフェースは極めて限られたものになります。次の Listing A.6 に示す通り、このベースクラスが提供しているメソッド群はすべて、インデックスの決定方法に関連するものです。

Listing A.6: Index クラスのインターフェース

```
class Index : public Observable {
  public:
    virtual ~Index() {}
    virtual std::string name() const = 0;
    virtual Calendar fixingCalendar() const = 0;
    virtual bool isValidFixingDate(const Date& fixingDate)
                                                  const = 0;
    virtual Real fixing(const Date& fixingDate,
                        bool forecastTodaysFixing = false)
                                                  const = 0;
    virtual void addFixing(const Date& fixingDate,
                           Real fixing,
                           bool forceOverwrite = false);
    void clearFixings();
};
```

isValidFixingDate() メソッドは、引数として取った日付が“インデックス決定日”に該当するか否か(あるいは将来そうなるのか)を示します。また、fixingCalendar() は、インデックス決定日を決めるカレンダーを返し、fixing() メソッドは、過去あるいは将来のインデックス決定日におけるインデックスの値を返します。その他のメソッドは、過去に決定したインデックス値に関するものです。name() メソッドは各インデックスを特定するインデックス名を返します。このメソッドは通常、保存されている様々な Index データから該当の Index データを取りだす為に使われます。addFixing() メソッドは、値が決定されたインデックス値を保存します(設定する値が複数でも、オーバーロード関数で対応しています)。また clearFixing() メソッドは、保存されているインデックスのデータを消去します。

インデックスのデータを保存・消去すると言ってますが、Index クラスのメンバー変数の中にデータを格納する変数が見当たりませんね?実は、過去のインデックス値は Index インスタンス毎に持つのでは無く、(その外に保存して)幅広くシェアして使いたいという要請に対応しました。仮に、6カ月 Euribor インデックスのインスタンスが、ある日付のインデックス決定値を格納しようとする場合、その値は、addFixing() を呼び出したインスタンスからだけではなく、同じ Index のすべてのインスタンスからも見えるようにしたいと考えました。

（注：ここで同じ Index のインスタンスとは、同じクラスの Index では無く、全く同一の Index のインスタンスを意味します。例えば `USDLibor(3*Months)` と `USDLibor(6*Months)` は全く同じインデックスとは見做しません。しかし、2つの異なる `USDLibor(3*Months)` インスタンスは、同一のインデックスインスタンスと見做します。）

その方法は、Singleton クラスの派生クラスとして `IndexManager` クラスを定義し、カーテンの後ろに隠しながら、それを使う事にしました。少し問題含みと思われますか？　確かにそうですが、それは、すべての Singleton クラスがそうだからです。代替策としては、インデックス値を格納する為に、各派生クラスに static なクラスをメンバー変数として持たせる方法があります（しかし、それは、Singleton と同様にデータの同一性原則に反します）。いずれにしても、この件については QuantLib ライブラリーの次の大きな改修時に再検討しなければなりません。（注：これまでの修正実績からすれば、次の大きなバージョンアップは2025年頃かも知れません。いいえ、冗談です。でもそうなるかも）

今の所、Index ベースクラスは Observer からは派生していません。しかし、いくつかの派生クラスではそうなっています（別に驚くにはあたりません。将来の Fixing（インデックス決定値）を予想する場合は、殆ど常に何等かの市場価格の Quote を参照するからです）。この点については、最初から決めた設計方法では無く、プログラムコードの開発を進めていくにつれて、自然とそうなって行ったもので、将来のバージョンで変更するかも知れません。しかし、Index ベースクラスを仮に Observer クラスから派生させたとしても、派生クラスにおいて発生する若干の重複を避ける事はできません。その理由については、もう少し詳しく説明する必要があるでしょう。

既に述べた通り、インデックスの fixing 値が更新されるケースは2通り考えられます。ひとつは、将来の fixing 値を予想する際に、データソースとなる Observable を参照する場合です。この場合、Index インスタンスは、自らをその Observable に登録すれば済む話です（これは、各派生クラスレベルで為されます。それぞれの派生クラスは、依存するデータソースが異なるからです）。もうひとつは、新しい fixing 値が決まって利用できる状態になった場合で、これは取り扱いが少し厄介になります。新しい fixing 値は、個別の Index インスタンスから `addFixing()` メソッドを呼び出す事によって、（`IndexManager` に）格納されます。そうすると、（`addFixing()` を呼び出したインスタンスにとっては）外部からデータ変更の通知は必要なさそうです。また、新しい fixing 値は（その Index インスタンスに登録されている）Observer インスタンスに通知するだけでよさそうです。しかし、そうではありません。既に述べた通り、fixing 値は共有されています。もし本日の3 Month Euribor の　fixing 値が決まり、それを（`IndexManager` に）格納したら、そのデータは、同じ3カ月 Euribor のすべてのインスタンスからも使える様になります。従って、そのすべてのインスタンスは、fixing 値が決まった事を知る必要があります。さらに、Instruments　や　TermStructure のインスタンスは、そういった個別の Index インスタンスに登録されているでしょうから、新しい Fixing 値が決まれば、それを知る必要があります。即ち、各 Index インスタンスは fixing 値が新しく決まった事を、それぞれに登録されているすべての Observer に通知しなければなりません。

これの解決策は、同じ種類の Index インスタンスは、Shared Object（具体的にはすべての fixing 値を格納している `IndexManager` クラスの Singleton）を経由して、お互いに、連絡を取り合えるようにする事です。`IndexManager` クラスは、各 fixing 値（の時系列データ）に、特定のインデックス名を示す名札（Tag）を付けます。そして、その Tag に対応する `ObservableValue` クラスのインスタンスを生成して、Tag と同一名の、fixing 値が加えられた場合、それを通知する仕組みを提供しています。（このクラスについては、後程この Appendix の中で説明しますので、ここでは詳細な説明

は必要ないでしょう）

これで、すべての部品が揃いました。Index インスタンスは、生成された後、IndexManager インスタンスに対し、name() メソッドで返されるインデックス Tag に対応する Shared Observable を要求します。その後、例えば 6month Euribor のインスタンスから addFixing() メソッドが呼び出された場合、新しい Fixing 値が IndexManager に格納され、かつ Shared Observable からすべての同じ Index インスタンス（6month Euribor インスタンス）に対して、通知が発せられます。それで、同じインデックスインスタンス全体がうまくいくという訳です。

しかし、C++ は、この歯車にモンキーレンチを入れて、邪魔しにきます。上記の通りであれば、Index のコンストラクターから次のようなメソッドを呼び出すだけで済ませようという誘惑にかられます。

```
registerWith(IndexManager::instance().notifier(name()));
```

しかし、この方法はうまく機能しません。理由は、ベースクラスのコンストラクターにより生成された仮想関数 name() メソッドは polymorphic ではないからです。

（注：C++ の問題の多い部分にあまり詳しくない読者の為に説明すると、ベースクラスのコンストラクターが呼び出された場合、その段階では未だ派生クラスのメンバー変数は生成されていません。個別の派生クラスに特有の動作が（訳注：ここでは派生クラスで実装される name() メソッド）、未だ生成されていない派生クラスのメンバー変数（個別の Index 名）に依存するので、C++ は派生クラスでの動作をあきらめ、ベースクラスで実装されている仮想関数を使おうとします）

この理由から、数段落前に指摘したプログラムコードの重複が発生します。すなわち、上記のメソッドの呼び出しは、派生クラスのコンストラクター毎に記述する必要があるのです。Index ベースクラスそのものは、実際の所（コンパイラーがデフォールトで提供しているもの以外は）コンストラクターを持っていません。

Index クラスの具体的な派生クラスでの実装例として、次の Listing A.7に InterestRateIndex クラスを示します。

Listing A.7: InterestRateIndex クラスの概要

```
class InterestRateIndex : public Index, public Observer {
  public:
    InterestRateIndex(const std::string& familyName,
                      const Period& tenor,
                      Natural settlementDays,
                      const Currency& currency,
                      const Calendar& fixingCalendar,
                      const DayCounter& dayCounter);
    : familyName_(familyName), tenor_(tenor), ... {
        registerWith(Settings::instance().evaluationDate());
        registerWith(
                 IndexManager::instance().notifier(name()));
    }
```

```
        std::string name() const;
        Calendar fixingCalendar() const;
        bool isValidFixingDate(const Date& fixingDate) const {
            return fixingCalendar().isBusinessDay(fixingDate);
        }
        Rate fixing(const Date& fixingDate,
                    bool forecastTodaysFixing = false) const;
        void update() { notifyObservers(); }

        std::string familyName() const;
        Period tenor() const;
        ... // other inspectors

        Date fixingDate(const Date& valueDate) const;
        virtual Date valueDate(const Date& fixingDate) const;
        virtual Date maturityDate(const Date& valueDate) const = 0;
      protected:
        virtual Rate forecastFixing(const Date& fixingDate)
                                                        const = 0;
        std::string familyName_;
        Period tenor_;
        Natural fixingDays_;
        Calendar fixingCalendar_;
        Currency currency_;
        DayCounter dayCounter_;
    };

    std::string InterestRateIndex::name() const {
        std::ostringstream out;
        out << familyName_;
        if (tenor_ == 1*Days) {
            if (fixingDays_==0) out << "ON";
            else if (fixingDays_==1) out << "TN";
            else if (fixingDays_==2) out << "SN";
            else out << io::short_period(tenor_);
        } else {
            out << io::short_period(tenor_);
        }
        out << " " << dayCounter_.name();
        return out.str();
    }
```

```
Rate InterestRateIndex::fixing(
                        const Date& d,
                        bool forecastTodaysFixing) const {
    QL_REQUIRE(isValidFixingDate(d), ...);
    Date today = Settings::instance().evaluationDate();
    if (d < today) {
        Rate pastFixing =
            IndexManager::instance().getHistory(name())[d];
        QL_REQUIRE(pastFixing != Null<Real>(), ...);
        return pastFixing;
    }
    if (d == today && !forecastTodaysFixing) {
        Rate pastFixing = ...;
        if (pastFixing != Null<Real>())
            return pastFixing;
    }
    return forecastFixing(d);
}

Date InterestRateIndex::valueDate(const Date& d) const {
    QL_REQUIRE(isValidFixingDate(d) ...);
    return fixingCalendar().advance(d, fixingDays_, Days);
}
```

予想された通り、このクラスは、Index ベースクラスから継承した動作以外に、かなりの数の個別の動作を定義しています。そもそも、このクラスは Observer クラスからも派生しています（Index ベースクラスはそうではありません）。InterestRateIndex クラスのコンストラクターは、金利インデックスを特定するのに必要なデータを、引数として取ります。すなわち、① “Euribor”といったファミリー名、② 同じファミリーの中の個別の期間（3か月とか6か月）、③ Fixing 日から決済日までの期間、④ 対象通貨、⑤ Fixing 日を決めるカレンダー、⑥（経過利息計算に必要な）日数計算方法などです。

引数で渡されたデータは、もちろん対応するメンバー変数に格納されます。そして、自分自身のインスタンスを、2つの Observable インスタンスに登録します。一つは、グローバル変数である EvaluationDate インスタンスです。これが必要な理由は、この Index インスタンスが、本日の Fixing 値を聞かれた時に起動される、“日付に関する特別な動作”に必要だからです。これについてはすぐ後で説明します。ふたつ目の Observable は、IndexManager の中に格納されているインスタンスで（訳注：先ほど説明された ObservableValue のこと）、新しい Fixing 値が決まった際、そこから各 Observer に通知が発せられます。この InterestRateIndex 派生クラスの階層で、もう Observable となる IndexManager 内のインスタンスを特定する事が可能です。InterestRateIndex クラスは金利インデックスを特定する情報をすべて持っており、このクラスで実装された name() メソッドを呼び出す事が出来ます。それは同時に、InterestRateIndex クラスからさらに派生するクラスで、name() メソッドをオーバーライドしてはいけない事を意味します。さ

らに派生するクラスでオーバーライドされたメソッドは、継承元のコンストラクターが呼び出された段階では呼び出す事は出来ない為（理由は少し前に説明しました）、その派生クラスが間違った `IndexManager` インスタンスに登録されてしまう事になります。残念ながら、C++ ではこれを防ぐ方法がありません（C++ では、Java の”final”や C# の“sealed”に相当する宣言文が無いからです）。代替策として、`InterestRateIndex` クラスから派生したクラスでも、必ず `IndexManager`（内の同一 `InterestRateIndex` を格納する Observable）に登録するメソッドを用意する方法です。しかし、それを強制する事はできませんし、エラーが起こりにくいものの、よりめんどくさくなります。

`InterestRateIndex` クラスで定義されている他のメソッドは、それぞれ独自の目的を持っています。いくつかは、ベースクラスである Index と Observer で宣言されたインターフェースを実装しています。最もシンプルなのは `update()` メソッドで、Observable からデータ更新の通知を受け取ると、それを（自分が登録している）他の Observer に転送するだけです。`fixingCalendar()` メソッドは、メンバー変数に保持されている Calendar インスタンスを返すだけで、`isValidFixingDate()` メソッドは、引数として渡された日付が Fixing 用の Calendar で営業日になっているかどうかチェックします。

`name()` メソッドは、少し複雑です。このメソッドは、インデックスのファミリー名、期間、日数計算方法の情報を繋ぎ合わせて、“Euribor 6M Act/360”とか、”USD Libor 3M Act/360”といった個別の金利インデックスの名前を作り、それを返します。Overnight、Tomorrow-Next、Spot-Next といった特殊な期間については、それぞれに対応する略称を使って、そのインデックス名を作ります。

`fixing()` メソッドが、このクラスのメインの動作を担います。まず、引数として渡された日付けが、Fixing 日に該当するか否かチェックし、該当しない場合は例外処理に移ります。次に、その日付けが本日と一致するかどうかチェックし、仮にそれが過去の日付の場合は、Fixing 値を Singleton である `IndexManager` インスタンスから探し、もしデータがそこに無ければ例外処理に移ります（過去の Fixing 値を予想する術は無いからです）。もし日付が本日の日付の場合、最初に `IndexManager` に Fixing 値を探しに行き、値があればそれを返しますが、そこに無い場合はまだ Fixing 値が決まっていないという事になります。その場合（あるいは日付が将来の日付の場合）、Index インスタンスは、Fixing の予想値を計算します。その計算は `forecastFixing()` メソッドを呼び出して行われますが、このメソッドは `InterestRateIndex` クラスの中では純粋仮想関数として宣言されているので、そこから継承する派生クラスで実装される必要があります。この `fixing()` メソッドが動作するロジックの為に、先ほど述べたように、このインスタンスを EvaluationDate インスタンスに登録する必要があったのです。`fixing()` の中での動作は、引数で渡された日が本日かどうかのチェックを含んでおり、日付が移った場合は、それを通知してもらう必要があるのです。

最後に、`InterestRateIndex` クラスは `Index` クラスから継承していないメソッドも幾つか定義しています。そのほとんどがインスペクター関数で、メンバー変数に保持されているインデックスのファミリー名や期間を返します。それ以外のメソッドは、主に日数計算を行っています。`valueDate()` メソッドは、Fixing 日に対応する金利期間のスタート日を返します（例えば、LIBOR 金利が適用になるロンドンインターバンク市場の預金は、大半の通貨において、Fixing 日の2営業日後がそれに該当します）。`maturityDate()` メソッドは、引数に Value Date（たった今説明したもの）を取り、金利期間の最終日（預金の最終日）を返します。`fixingDate()` メソッドは `valueDate()` メソッドの逆関数となる動作で、Value Date を引数で取り、Fixing 日を返します。これらのメソッドの内のいくつかは、仮想関数であり、派生クラスでオーバーライド出来ます。例えば、`valueDate()` メソッドのデフォールトの動作は、メンバー変数に持つ Fixing Days（Fixing 日から Value Date ま

での日数）の情報と適用になるカレンダーを基に、その営業日数分だけカレンダーを前に進めた日を返します。しかし、LIBOR インデックスでは、まず London のカレンダーを使ってそれを行った後、インデックスの対象通貨を基に、その通貨に対応するカレンダーを使って再調整をする必要があるので、通貨によっては、その部分を派生クラスでオーバーライドする必要があります。ところが、何らかの理由で、`fixingDate()` メソッドは仮想関数として宣言されていませんでした。おそらく、見逃していたのだと思いますが、将来リリースするバージョンで修正しなければなりません。

Aside: どの程度まで汎用化する必要があるのか？

`InterestRateIndex` クラスのいくつかのメソッドは、明らかに、LIBOR インデックスを想定して作られています。なぜなら、QuantLib ライブラリーで最初に作られた Index クラスが LIBOR インデックスだったからです。その為、このクラスの汎用性が若干犠牲になっています。例えば、5年と10年のスワップ金利の金利差を金利インデックスとして取り扱おうとした場合、それをベースクラスのインターフェースや、`tenor()` のようなメソッドにフィットさせるのは大変な作業になるでしょう。しかし、一方で、そのような例を具体的に実装して試してみる事なく、Index クラスのインターフェースをより汎用的に書き換えるのは、賢明ではありません。2つの金利インデックスの“スプレッド”は（単なる金利差であり、インデックスでは無いと考えられるので）、Index クラスの汎用化の対象とすべきではないでしょう。

11.3.4 Exercise クラスと Payoff クラス

このセクションの最後に、特定の Instruments クラスの定義の中で使われているその商品用の特別なクラスについて説明します。

最初に、下記 Listing A.8に示す `Exercise` クラスからです。

Listing A.8: `Exercise` クラス、およびその派生クラスのインターフェース

```
class Exercise {
  public:
    enum Type {
        American, Bermudan, European
    };
    explicit Exercise(Type type);
    virtual ~Exercise();
    Type type() const;
    Date date(Size index) const;
    const std::vector<Date>& dates() const;
    Date lastDate() const;
  protected:
    std::vector<Date> dates_;
    Type type_;
```

```
};

class EarlyExercise : public Exercise {
  public:
    EarlyExercise(Type type,
                  bool payoffAtExpiry = false);
    bool payoffAtExpiry() const;
};

class AmericanExercise : public EarlyExercise {
  public:
    AmericanExercise(const Date& earliestDate,
                     const Date& latestDate,
                     bool payoffAtExpiry = false);
};

class BermudanExercise : public EarlyExercise {
  public:
    BermudanExercise(const std::vector<Date>& dates,
                     bool payoffAtExpiry = false);
};

class EuropeanExercise : public Exercise {
  public:
    EuropeanExercise(const Date& date);
};
```

皆さん予想されていたでしょうが、ベースクラスは、行使日の情報を取り出すメソッドを宣言しています。見ての通り、そのメソッドは複数あります。`dates()` メソッドは、行使日の配列全体を返します。また `date()` メソッドは、行使日の配列から、引数で指定されたインデックスに該当する行使日を返します。便利なメソッドである `lastDate()` は、ご想像の通り、最終行使日を返します。従って、若干の機能の重複があります。(注:カプセル化が好きな人にとっては、行使日の配列を返すメソッドより、特定の配列インデックスを指定してそれに該当する行使日を返すメソッドの方を好まれるでしょう。配列を返すメソッドは、カプセル化されたデータの中から必要以上の情報を外に出すことになるからです。)

また、`type()` メソッドが定義されていますが、それをみて、ちょっと頭をかかえてしまいました。このメソッドは、クラス内で宣言された、Type という名前の enum リスト(具体的には、European、Bermudan、American という行使タイプ)から、どれかをピックアップするものです。それ自体はおかしくありません。しかし、その次にあるクラス群の宣言は、このメソッドの趣旨に反するものです。すなわち、このクラスから派生させて、`AmericanExercise`、`BermudanExercise`、`EuropeanExercise` クラスを宣言してしまいました。Exercise ベースクラスで仮想デストラクターを宣言しているのを見ると、新しい行使タイプを定義したい場合は、継承を使う事を想定していたのでしょう。しかし一方

で、ベースクラスで行使 Type を enum で宣言したのは、その想定に反しています。なぜなら、新しい行使タイプを派生クラスで追加した場合、ベースクラスの enum リストも追加しなければならないからです。継承を想定していたもうひとつの証拠は、QuantLib の中のあちこちで使われているイデオムを見ると、Exercise ベースクラスのインスタンスを生成し、そのスマートポインターを使い回している事です。一方で、継承の想定に反しているのは、ベースクラスでは、デストラクター以外は仮想関数を宣言しておらず、派生クラスで追加される動作（メソッド）は、すべてベースクラスのメンバー変数を使っている事です。要するに、このコードを書いた当時、我々は（現在の私よりも）もっと頭の中が混乱していたのでしょう。

もしこれを今書き直すとしたら、おそらく enum リストを残して、Exercise ベースクラス自体を具体的なクラス（訳注：仮想関数を持たず、それ自体のインスタンスを生成して使えるクラス）にしたでしょう。その場合、派生クラスを作ったとしても、それはベースクラスのインスタンスを安全に切り出して使い回しができるようなオブジェクトにするか、あるいは Exercise インスタンスを直接返すような関数オブジェクトにするかでしょう。こうすると、オブジェクト指向を信奉するプログラマーを不機嫌にさせるかも知れません。しかしその方が、QuantLib のあちこちにある行使タイプをチェックする場所で、型変換や Visitor パターンを使わずに enum リストを使うだけで済むので、現実的です。派生クラスで、特別な動作を実装していないのを見ても、これでいいのではないかと思います。

この Section を書いていて、“オプション行使”は、Chapter IV で解説した `Event` クラスに含めても良かったのではないかと思えてきました。（訳注：オプション行使というのは、新たに発生する経済事象（Event）なので、Event クラスの概念にマッチしそうに見える。）しかし、Exercise クラスがオブジェクトモデル化している概念は、必ずしもそれとマッチしません。ヨーロピアンタイプの Exercise であれば、それを Event クラスのインスタンスとしてもいいでしょう。バーミューダタイプの Exercise であれば、Event の配列とすればいいでしょう。しかし、アメリカンタイプでは、行使日の“範囲”の概念をモデル化しなければなりません。そうすると、Exercise クラスで用意されているインターフェースの意味も変わってきます。なぜなら、`dates()` メソッドは、行使可能日の配列を返すのではなく、単に行使可能期間の最初の日と最後の日を返すだけです。よく起こる事ですが、一見簡単そうにみえても、注意深く見てみるとモデル化が難しいケースがあるものです。

*　　*　　*

次に、Payoff クラスを見てみます。下記 Listing A.9に、そのインターフェースといくつかの派生クラスを示します。

Listing A.9: Interface of the `Payoff` class and a few derived classes.

```
class Payoff : std::unary_function<Real,Real> {
  public:
    virtual ~Payoff() {}
    virtual std::string name() const = 0;
    virtual std::string description() const = 0;
    virtual Real operator()(Real price) const = 0;
    virtual void accept(AcyclicVisitor&);
};

class TypePayoff : public Payoff {
  public:
    Option::Type optionType() const;
  protected:
    TypePayoff(Option::Type type);
};

class FloatingTypePayoff : public TypePayoff {
  public:
    FloatingTypePayoff(Option::Type type);
    Real operator()(Real price) const;
    // more Payoff interface
};

class StrikedTypePayoff : public TypePayoff {
  public:
    Real strike() const;
    // more Payoff interface
  protected:
    StrikedTypePayoff(Option::Type type,
                      Real strike);
};

class PlainVanillaPayoff : public StrikedTypePayoff {
  public:
    PlainVanillaPayoff(Option::Type type,
                       Real strike);
    Real operator()(Real price) const;
    // more Payoff interface
};
```

このクラスのインターフェースの内、`operator()` は、引数として対象資産価格を受取り、それに

対応する行使価値 (payoff) を返します。また `accept()` は、Visitor パターン用のメソッドです。その他にインスペクター関数が2種類あり（`name()` と `description()` メソッド）いずれもレポート用に名前を返しますが、片方は余分でした。

開発時にこのクラスをモデル化するにあたって、我々はこのオブジェクトの使用方法について十分把握できていませんでした。その結果、残念なことに、出来上がったインターフェースは、あまり応用が利きません。最も大きな問題は、`operator()` メソッドが、引数として1つの対象資産価格しか取れないという点です。（注：この点は、対象資産が1つであっても、複数の Fixing 値で Payoff が決まる場合でも問題となります。）

もうひとつの大きな問題は、“継承”に頼りすぎている事です。例えば、Payoff の派生クラスとして `TypePayoff` を作り、そこで行使タイプと（Call または Put を指定していますが、それだけだと制約的かも知れません）、それに対応するインスペクター関数を加えています。さらにそこから `StrikedTypePayoff` クラスを派生させ、行使価格の情報を加えて、そこからさらに `PlainVanillaPayoff` クラスを派生させ、最終的にそれがシンプルな Call または Put の Payoff の具体的クラスになります。ここまでで、ベースクラスから3段階もの階層になっており、シンプルな `Payoff` 用のオブジェクトモデルとしては、階層化のやりすぎでしょう。今から QuantLib を使って、教科書的なオプションのプログラムコードを学ぼうとしている人にとっては、非常に解りにくくなっています。

もうひとつ我々がやった失敗は、Option クラスに Payoff インスタンスへの“ポインター”をメンバー変数として持たせた事です。当然、そのポインター（が示すインスタンス）には Payoff の情報が含まれているものと想定しての事です。その結果、`FloatingTypePayoff` クラスのようなもの（上記 Listing に記述されています）が出来上がってしまいました。このクラスは、Floating Look-Back Option の実装で使われ、オプションのタイプ（Call か Put か）の情報を保持します。しかし、このオプションの行使価格は、オプション満期時に決まるので、その時点まで Payoff（行使価値）を計算できず、従って、`payoff()` インターフェースを実装する事が出来ません。その結果、我々はこのクラスの `operator()` メソッドが呼び出された場合、例外処理に飛ぶようにしました。この場合、あえて payoff 値を計算させず、ルックバックオプションというタイプ名を返す事もできたでしょう。それは即ち、ベースのオプションクラスが、この時点での payoff 値を期待していない場合だけ可能です。次に、プログラムコードを見直す時に（改良すべき点として）覚えておかなければなりません。

11.4 数値計算関連のクラス群

QuantLib ライブラリーは、C++ の標準ライブラリーが提供する算術計算ツール以外に、いくつかのツールを用意する必要があります。ここでは、それらについて概略を説明します。

11.4.1 Interpolation：補間関数

Interpolation に関する関数群は、QuantLib の中では珍しく使用するのがやや危険なクラスです。

ベースクラスである Interpolation クラスの概要を下記 Listing A-9 に示します。このクラスは、x と y の2つのデータ配列を使って、x軸上の任意の点に対応するyの値を線形補間して導出しま

す(訳注:イールドカーブで言えばxが期間、yがその期間に対応するイールド)。このクラスは、その補間された値を返す `operator()` メソッドや、いくつかの便利な関数を用意しています。少し前に説明した Calendar クラスと同様、このクラスも pimpl idiom(Pointer to Implementation)を使って polymorphic な動作を実現しています。すなわち、まず内部クラスとして Impl クラスを宣言し、その Impl の派生クラスが個別の Interpolation アルゴリズムを実装します。そして Interpolation ベースクラスのメソッドから Impl 派生クラスのメソッドを呼び出す形でそれを使います。もうひとつの内部クラスである templateImpl クラスは、一般的な動作を実装すると供に、Interpolation の対象となるデータを格納します。

Listing A.10:`Interpolation` クラスの概要

```
class Interpolation : public Extrapolator {
  protected:
    class Impl {
      public:
        virtual ~Impl() {}
        virtual void update() = 0;
        virtual Real xMin() const = 0;
        virtual Real xMax() const = 0;
        virtual Real value(Real) const = 0;
        virtual Real primitive(Real) const = 0;
        virtual Real derivative(Real) const = 0;
    };
    template <class I1, class I2>
    class templateImpl : public Impl {
      public:
        templateImpl(const I1& xBegin, const I1& xEnd,
                     const I2& yBegin);
        Real xMin() const;
        Real xMax() const;
      protected:
        Size locate(Real x) const;
        I1 xBegin_, xEnd_;
        I2 yBegin_;
    };
    boost::shared_ptr<Impl> impl_;
  public:
    typedef Real argument_type;
    typedef Real result_type;
    bool empty() const { return !impl_; }
    Real operator()(Real x, bool extrapolate = false) const {
        checkRange(x,extrapolate);
        return impl_->value(x);
    }
```

```
    Real primitive(Real x, bool extrapolate = false) const;
    Real derivative(Real x, bool extrapolate = false) const;
    Real xMin() const;
    Real xMax() const;
    void update();
  protected:
    void checkRange(Real x, bool extrapolate) const;
};
```

コードを見ての通り、`templateImpl` クラスは（対象データとなる）x と y の値をコピーしていません。その代わり、この2種類の変数列に対する Iterator（ポインター）を保持して、データ値を見に行けるようにしています。これが、Interpolation クラスを使うのが安全でない理由です。（危険を回避するには）まず、Interpolation インスタンスの生存期間が、対象となるデータ変数列のそれを越えないようにしなければなりません。すなわち、メモリーから対象データを消去した場合、(Interpolation インスタンスが持つ) そこへのポインターを残してはいけないという事です。それと同時に、Interpolation インスタンス自体を保持するオブジェクトをコピーする際は、細心の注意を払わなければなりません。

前者の注意点は、それほど大きな問題ではありません。Interpolation インスタンスが単独で使われる事は殆どありません。通常は、他のクラスのメンバー変数として、Interpolation の対象データと一緒に格納されます。その結果、Interpolation インスタンスと対象データは（それを保持するオブジェクトの生成・消去の際に）同時に生成され、同時に消去されるので、生存期間の問題は、保持するオブジェクトが（自動的に）管理します。

後者の注意点も、さほど大きな問題ではありませんが、前者の問題は、通常、自動的に管理されるのに対し、後者の問題はユーザーが注意する必要があります。今述べた通り、`Interpolation` インスタンスは、通常、他のオブジェクトのメンバー変数として対象データと一緒に保持されます。そのオブジェクト用にコンパイラーが生成するコピーコンストラクターは、保持しているメンバー変数の新しいコピーも生成します。対象データについてはそれでかまいません。しかし Interpolation インスタンスの新しいコピーは、引き続きコピー元の対象データへのポインターを持つことになります（なぜなら Interpolation インスタンスのコピーは、元の対象データへの Iterator のコピーだからです）。この動作は、当然正しくありません。

これを避ける為には、ホストのクラス（コンテナーとして Interpolation インスタンスと対象データをメンバー変数に持つクラス）の開発者は、ユーザー定義のコピーコンストラクターを作る必要があり、そこではメンバー変数をコピーするだけでなく、新たな Interpolation インスタンスを生成して、それがコピーされた対象データへのポインターを保持するようにしなければなりません。しかし、それは単純な操作ではありません。Interpolation インスタンスを保持するコンテナーオブジェクトは、その正確なデータ型の情報を持っていない（それは `Impl` クラスの中に隠されている）ので、新しい Interpolation インスタンスを再生成できないのです。

この問題の解決方法のひとつは、Interpolation クラスに、仮想関数として、自分と同じ型のインスタンスを生成する `clone()` メソッドのようなものを持たせるか、コピーされた際に、対象データへのポインターをコピーされた方の対象データへのポインターに変更する `rebind()` のようなメ

ソッドを持たせる事です。しかし、あえてそうする必要はありませんでした。なぜなら、Interpolation Traits が既に用意されて、殆どの場合これで対応出来るからです。

何それ?と言われますよね。実は、Chapter III で、TermStructure クラスの Interpolation の概要を説明した際に出てきた `Linear` や `LogLinear` クラスの事です。そのひとつの例を、次の Listing A.11に示します。

Listing A.11:`LinearInterpolation` クラスと、その traits クラスの概要

```
template <class I1, class I2>
class LinearInterpolationImpl
    : public Interpolation::templateImpl<I1,I2> {
  public:
    LinearInterpolationImpl(const I1& xBegin, const I1& xEnd,
                            const I2& yBegin)
    : Interpolation::templateImpl<I1,I2>(xBegin, xEnd, yBegin),
      primitiveConst_(xEnd-xBegin), s_(xEnd-xBegin) {}
    void update();
    Real value(Real x) const {
        Size i = this->locate(x);
        return this->yBegin_[i] + (x-this->xBegin_[i])*s_[i];
    }
    Real primitive(Real x) const;
    Real derivative(Real x) const;
  private:
    std::vector<Real> primitiveConst_, s_;
};

class LinearInterpolation : public Interpolation {
  public:
    template <class I1, class I2>
    LinearInterpolation(const I1& xBegin, const I1& xEnd,
                        const I2& yBegin) {
        impl_ = shared_ptr<Interpolation::Impl>(
            new LinearInterpolationImpl<I1,I2>(xBegin, xEnd,
                                               yBegin));
        impl_->update();
    }
};

class Linear {
  public:
    template <class I1, class I2>
    Interpolation interpolate(const I1& xBegin, const I1& xEnd,
                              const I2& yBegin) const {
```

```
            return LinearInterpolation(xBegin, xEnd, yBegin);
        }
        static const bool global = false;
        static const Size requiredPoints = 2;
    };
```

LinearInterpolation クラスの構造は単純です。このクラスは template 引数を持つコンストラクターを定義し（但しこのクラスそのものは Template ではありません）、そこで自分用の Impl クラス（LinearInterpolationImpl クラス）のインスタンスを生成します。LinearInterpolationImpl クラスは、templateImpl クラスから継承されており、実際の Interpolation アルゴリズムの実行という重たい作業を行います（その時、ベースクラスで定義されている locate() クラスの助けを借りています）。

Linear traits クラス（上記コード中の Linear クラス）は、2つの静的変数を定義しています。すなわち、Interpolation に最低限必要なデータ数が2である事（requiredPoints=2）、及びある点における値を変化させた時の影響が局所的である事という事（global=false）です。また interpolate() メソッドを定義しており、引数で取った Iterator x と y を使って、この traits に対応する型の Interpolation インスタンスを生成します。このメソッドは、すべての traits で、同じインターフェース（引数のセット）を使って実装されており（仮に Spline 関数のように Interpolation により多くのパラメータを必要とする場合は、それらの情報は traits のコンストラクターに渡されそこで保持されます）、先ほどのコピーコンストラクターで発生する問題の解決策も提供しています。Chapter III で解説した InterpolatedZeroCurve クラスのコードを見て頂くと、そこでは traits クラスのインスタンス（そこで interpolator_と呼ばれる変数です）の他に、Interpolation インスタンスと対象データも一緒に保持しているのが判ると思います。Interpolation インスタンスを保持するクラスについて、すべて同じ様な設計にすれば、（ユーザーが定義する）コピーコンストラクターの中で、traits に対応する型の Interpolation インスタンスを（対象データと同時に）生成出来ます。

残念ながら、Interpolation インスタンスを保持するクラスで、このようなコピーコンストラクターをユーザーが定義する事を強制する術がありません。従って、開発者はそれを覚えておかなければなりません。Interpolation クラスをコピー不可にするか、コピーを防ぐ何等かの idiom（訳注：プログラム言語に備わる構文上のルール）が無い限り、これを防ぐ方法はありません。C++11では、コピー不可かつ movable に出来るので、対応可能です。

Aside: ゴルディオスの結び目（一見複雑に見える問題をいとも簡単に解く方法）

Interpolation インスタンスを保持するクラスを実装する際、ユーザー定義のコピーコンストラクターを作る以外に問題を防ぐ方法は、そのクラスをコピー不可にする事です。それは、見た目ほど問題ではありません。Interpolation インスタンスを保持するオブジェクトは、通常 TermStructure クラスが想定されますが、大半のケースで、このクラスのインスタンスは shared_ptr を使ってやり取りされるので、コピーコンストラクターの出る幕が無いのです。（本当の話、殆どのカーブのクラスは Release 1.0が出るまではコピー不可になっていましたが、だれもそれについて苦情を言っ

てませんでした。結果的に、便宜上コピー可に戻しましたが、そうする必要があったのか、未だ確信はありません。）

最後の注意点です。Interpolation インスタンスは、対象データに対する Iterator を保持していますが、それだけではそのデータが更新された際のアップデートの仕組みとしては不十分です。対象データの更新があった場合、update( ) メソッドが呼び出され Interpolation も更新できるようにするべきですが、それは（今の設計の仕組みでは）Interpolation インスタンスと対象データを保持しているクラスの役割です（但し、対象データの方は、おそらく何等かのデータソースに対し Observer として登録されているでしょう）。しかし Observer としての登録は、データを直接読み込んでいる Interpolation オブジェクトにも当てはまります。クラスによって実装の違いはありますが、LinearInterpolation のように、いくつかの事前計算を行い、そのデータを自分で格納しているクラスもあります。`LinearInterpolation` クラスの実装では、ポイント間の傾きと各ポイントの原データを、（事前計算して）格納する様になっています。対象データがどの程度頻繁に更新されるかによりますが、この事前計算の仕組みは、パフォーマンス上、最適かも知れないし、あるいは、全くその逆の可能性もあります。

（注：このクラスにある `primitive()` と `derivative()` メソッドは、若干 Implementation Leak（訳注：本来ならベースクラスで一般化したメソッドとして定義、実装されるべきものが、そこで漏れた為、派生クラスにおいてメソッドを定義・実装している）と言えるかも知れません。このメソッドは Interpolate されたイールドカーブにおいて、ゼロ金利とフォワード金利の間の換算をする時に使われます。（訳注：従って、`LinearInterpolation` に限らず Interpolation クラス一般で使えるメソッドとした方が良かったかも知れない））

11.4.2 One-dimensional Solvers: 1 次元ソルバー

ソルバーは、Chapter III で説明した Bootstrapping の計算ルーティンや、Chapter IV で説明した利回りの計算などの様に、計算値を特定のターゲット値に収束させる計算アルゴリズムの中で使われます。すなわち、ある関数 $f(x)$ が与えられ、$f(x) = \xi$ が所定の誤差内に留まるような x を求めるアルゴリズムです。

既存のソルバーは、$f(x) = 0$ となる様な x を求めるアルゴリズムです。もちろん、これによって（訳注：ターゲット値を ξ ではなく 0 にする事によって）汎用性を失うものではありません。但しその場合は、ユーザーが追加のヘルパー関数　$g(x) \equiv f(x) - \xi$ を定義する必要があります。QuantLib ライブラリーには、いくつかの（1次元）ソルバーが用意されていますが、すべて Numerical Recipes in C (Press *et al*, 1992) から取ってきて、それを書き換えて実装したものです。

（注：いくつかの多次元 Optimizer も同じ本から取ってきています。著作権の問題と同時に、C++ の文法に適合させる必要もあり、若干書き換えています（例えば、配列のインデックスは 0 からスタートする）。）

下記の Listing A.12 は、複数のソルバーの土台として使われている Solver1D クラステンプレートのインターフェースを示しています。

Listing A.12 : `Solver1D` クラスのインターフェースと、いくつかの派生クラス

```
template <class Impl>
class Solver1D : public CuriouslyRecurringTemplate<Impl> {
  public:
    template <class F>
    Real solve(const F& f,
               Real accuracy,
               Real guess,
               Real step) const;
    template <class F>
    Real solve(const F& f,
               Real accuracy,
               Real guess,
               Real xMin,
               Real xMax) const;
    void setMaxEvaluations(Size evaluations);
    void setLowerBound(Real lowerBound);
    void setUpperBound(Real upperBound);
};

class Brent : public Solver1D<Brent> {
  public:
    template <class F>
    Real solveImpl(const F& f,
                   Real xAccuracy) const;
};

class Newton : public Solver1D<Newton> {
  public:
    template <class F>
    Real solveImpl(const F& f,
                   Real xAccuracy) const;
};
```

このクラスは、すべてのソルバーに共通して使える、複数の同一名のメソッドを提供しています。オーバーロードされた `solve()` メソッドの内、ひとつは解を含むレンジの上限と下限を探します。もう一つの `solve()` メソッドは、引数として渡された上限と下限の間に解が存在するかどうかチェックします。いずれのメソッドも、実際の計算アルゴリズムは、派生クラスで実装されている `solveImpl()` メソッドに委託されています。その他のメソッドは、求める解の範囲に関する‘条件’を課したり、関数の（最大）計算回数を設定したりします。

実際の計算アルゴリズムを `solveImpl()` に委託するやり方は、Curiously Recurring Template Pattern（“CRTP”）を使って実装されています。この Pattern については Chapter VII で既に説明し

ています。ソルバーのプログラムを書いていた当時、テンプレートクラスをしきりに使おうとしていました。(その頃 expression templates (Veldhuizen, 2000) を実装しようとした事は、既に述べましたっけ?)従って、この Pattern を選択したのは、当時の流行に影響されたと思われるかも知れません。しかし、dynamic polymorphism を使おうとすれば、それ以外に選択肢はありませんでした。われわれは、ソルバーを、関数ポインターや関数オブジェクトと一緒に機能させたいと考えていました。またその頃は、`boost::function` は存在していませんでした。その結果、template メソッドを使うに至った訳です。template メソッドは仮想関数とする事が出来ないので、CRTP が唯一、共通のメソッドをベースクラスに置き、そこから派生クラスで定義・実装されたメソッドを呼び出す方法でした。

このセクションを終わるにあたっていくつかの注意点を述べます。一点目は、CRTP を使っている為、ユーザーがソルバーを使う関数をプログラムする場合は、テンプレート関数にする必要があります。これは少し変に感じるかも知れません。実際には、我々がプログラムしたほとんどのケースで、そんな事を気にせずに、ソルバーを最初に明示的に選択して対応していました。ユーザーの方が同じ事を行っても非難するつもりはありません。二点目は、ほとんどのソルバーは、引数として渡された関数 `f` を呼び出して `f(x)` を計算するだけなので、それがどんな関数であってもソルバーは動作します。しかし `Newton` クラスや `NewtonSafe` クラスでは、`f.derivative(x)` を定義する必要があります(訳注:ニュートン法は微分係数を使って、収束速度を速めてるアルゴリズムなので、微分不可能な関数では対応できない)。これもまた、動的な polymorphism を使っているのに、変に感じるかも知れません。三点目は、`Solver1D` のインターフェースを見ても、引数として渡される誤差の許容値 ϵ が、x の許容誤差(推定値 $\tilde{x}$ と真の解 x との許容誤差)を意味するのか $f(x)$ の許容誤差($f(\tilde{x})$ の計算結果が 0 からどれだけ離れてもよいか)を意味するのか明確ではありません。これについては、既存のソルバーはすべて、x の許容誤差として使っています。

11.4.3 Optimizers: 多次元関数の最小値問題

多次元の Optimizer は1次元のソルバーより、ずっと複雑です。Optimizer の役割を掻い摘んで言うと、Cost Function(コスト関数または目的関数)$f(\mathbf{x})$ が最小値を返すような変数ベクトル $\tilde{\mathbf{x}}$ を探し出す事ですが、その説明だけでは不十分です。それについては Cost Function の実装内容を見ていく際に、もう少し詳しく説明します。

Optimizer のオブジェクトモデル化においては、(Solver の時とは異なり)テンプレートを使いませんでした。すべての Optimizer クラスは、次の Listing A.13に示す `OptimizationMethod` ベースクラスから派生しています。

Listing A.13:`OptimizationMethod` クラスのインターフェース

```
class OptimizationMethod {
  public:
    virtual ~OptimizationMethod() {}
    virtual EndCriteria::Type minimize(
                    Problem& P,
                    const EndCriteria& endCriteria) = 0;
};
```

このクラスが提供しているメソッドは、デストラクター以外は `minimize()` メソッドのみです。このメソッドは引数として `Problem` クラスのインスタンスを取ります。そのインスタンスが、最小値を求める“関数インスタンス”とその“制約条件インスタンス”への参照を持ち、そこで実際の計算アルゴリズムが実行されます。計算が終了すると、多数の計算結果を返す `Problem` インスタンスが出来あがります。`Problem` インスタンスは計算結果として、最適解となるベクトル $\tilde{\mathbf{x}}$ の他に、計算がどのように終了したか、その理由も返さなければなりません（最小値を探す計算がうまく収束したのか、それとも計算反復回数の最大値に到達したのでその時点での推定値を返すのか）。さらに、導出された最適解 $\tilde{\mathbf{x}}$ を使った関数値そのものを返す事が出来れば、もっと良かったかもしれません（既に計算されているはずですから）。

現時点での実装は、このメソッドは計算が終了した理由を返すのみで、実際の計算結果は `Problem` インスタンスの中に保存されたままです（メソッドの戻り値として返さない）。これが、メソッドの引数である Problem インタンスが、non-const な“参照”として渡されている理由です。別の実装方法として、Problem インスタンスはそのままにして、計算結果をすべて“構造体”に放り込んで、それを返す事も出来きましたが、その方がより面倒くさくなると思います。一方で（引数の Problem インスタンスだけでなく）`minimize()` メソッド自体も non-const にした理由は良く分かりません。おそらく、見逃してしまったのでしょう（これについては、後程振り返ります）。

次に、下記 Listing A.14 に示す `Problem` クラスの説明に進みます。

Listing A.14:`Problem` クラスのインターフェース

```
class Problem {
  public:
    Problem(CostFunction& costFunction,
            Constraint& constraint,
            const Array& initialValue = Array());

    Real value(const Array& x);
    Disposable<Array> values(const Array& x);
    void gradient(Array& grad_f, const Array& x);
    // ... other calculations ...

    Constraint& constraint() const;
    // ... other inspectors ...
```

```
        const Array& currentValue();
        Real functionValue() const;
        void setCurrentValue(const Array& currentValue);
        Integer functionEvaluation() const;
        // ... other results ...
    };
```

既に述べたように、このクラスは、最小値を求める Cost Function（目的関数）や、制約条件や、適当な推定値、といった、最小値問題に必要な引数を取り纏めたクラスです。また、対象となる Cost Function を呼び出しながら、同時に試行回数のカウントを行うメソッドも提供しています（それについては、Cost Function の説明をする時に詳しく解説します）。さらに、いくつかのインスペクター関数や計算結果を取りだすメソッドも提供しています。（Optimizer の中で使われる、それらの値を設定するメソッドもあります）。

このクラスの問題点は、そういった計算で使われる要素を、non-const な参照として取り込み、保存している事です。それらを const として取り扱うかどうかについては、後ほど説明します。それらが“参照”であること自体が問題です。なぜなら、これらのインスタンスの生成消去を少なくとも Problem インスタンスの生成消滅に合わせる責任を、ユーザー側の方で負わなければならないからです。

代替策として Optimizer が、計算結果を構造体で返すようにする方法を取っても、若干問題は残ります（訳注：先ほどの説明に合った通り、今の実装方法は計算結果を Problem インスタンスに書き込んでいる）。もしそうするなら、Problem クラスを取り除いて、計算で使う要素を、直接 Optimizer クラスの minimize() メソッドに渡せばいいでしょう。こうすれば、先ほどの non-const の”参照”をメモリーに保持しない為、メモリー領域の生成消去の問題は回避できます。しかし、その場合は、各 Optimizer が Cost Function の試行回数のカウントを行う必要があり、プログラムコードを重複させる原因になります。

また、1次元ソルバーとは異なり、Cost Function は minimize() メソッドのテンプレート引数になっていません。各 Cost Function は次の Listing A.15にある CostFunction ベースクラスから継承する必要があります。

Listing A.15：CostFunction クラスのインターフェース

```
    class CostFunction {
      public:
        virtual ~CostFunction() {}

        virtual Real value(const Array& x) const = 0;
        virtual Array values(const Array& x) const = 0;

        virtual void gradient(Array& grad, const Array& x) const;
        virtual Real valueAndGradient(Array& grad,
                                      const Array& x) const;
```

```
        virtual void jacobian(Matrix &jac, const Array &x) const;
        virtual Array valuesAndJacobian(Matrix &jac,
                                        const Array &x) const;
    };
```

当然ですが、このクラスのインターフェースで value() メソッドを宣言しており、このメソッドは引数として与えられた変数の配列を使って、関数の計算結果を返します（読者の方は、ここで operator() メソッドを宣言していない事に驚いたかも知れません）。一方で、values() メソッドも宣言されており、これは配列を返します。Optimizer が、複数の Quote データを使って Calibrate するのに使われる事を思い出せば、それ程驚く事ではないでしょう。value() メソッドが誤差値の合計（すなわち変数毎の誤差の2乗平均、あるいは同様の数値）を返すのに対し、values() メソッドは、（入力された変数である）Quote 毎の誤差値の配列を返します。この数値は、Optimizer の収束速度を向上させるようなアルゴリズムの中で使われます。

その他のメソッド群は、各変数に対する微分を返し、その値は特別な計算アルゴリズムの中で使われます。その内の一つ gradient() メソッドは各変数に対する微分を計算し、その値を第一引数として渡された配列（上記コードでは Array& grad）に保持します。jacobian() メソッドは、同じ操作を values() メソッドの為に行い、計算値を行列に保持します。さらに valueAndGradient() と valuesAndJacobian() メソッドは、計算誤差の値と変数に対する微分の計算の両方を同時に行います。この2つのメソッドはデフォールトとして有限差分による微分を計算しています。当然ながら相応の計算時間がかかります。仮に解析的な方法で、これらのメソッドをオーバーライドできないのであれば、差分を使った方法を使う価値があるかどうか確信が持てません。

ひとつ注意点があります。CostFunction のインターフェースをチェックして判った事ですが、すべてのメソッドが const として宣言されています。従って、このクラスのインスタンスを Problem クラスのコンストラクターに non-const な参照として渡す事は、間抜けなやり方でした。それを const に変更するのはコンストラクターの役割を広げるだけなので、過去のバージョンとの互換性を壊す事なく変更可能でしょう。

最後に、Constraint クラスを下記 Listing A.16に示します。

Listing A.16: Constraint クラスのインターフェース

```
    class Constraint {
      protected:
        class Impl;
      public:
        bool test(const Array& p) const;
        Array upperBound(const Array& params) const;
        Array lowerBound(const Array& params) const;
        Real update(Array& p,
                    const Array& direction,
                    Real beta);
        Constraint(const shared_ptr<Impl>& impl =
                                          shared_ptr<Impl>());
    };
```

このクラスは Cost Function の定義域に適用される“制約条件”のベースクラスです。QuantLib ライブラリーは、ここでは示していませんが、いくつかの定義済みのクラスと、それらをひとつに纏めた `CompositeConstraint` クラスを定義しています。

このクラスの中心となるメソッドは test( ) で、変数の配列を引数として取り、それらの値が制約条件を満たしているかどうかをチェックします。すなわち、配列の値が変数の領域内であれば、有効とします。このクラスは、`upperBound( )` と `lowerBound( )` メソッドを定義しており、理屈では変数の上限と下限を設定することになっていますが、実際にはそれを常に正しく設定できません。例えば領域が円であった場合、ある点のx座標と y 座標が、それぞれ上限の内側にあったとしても、その点そのものは円領域の外という事が起こり得ます。

さらに注意点を2つ。ひとつ目は、`Constraint` クラスは `update( )` メソッドを定義していますが、`const` として宣言されていません。もしこのメソッドの目的が制約条件自体を変更するなら、そうする意味があったでしょうが、そうでないなら問題です。実際には、このメソッドは変数の配列とその方向の配列を引数として取り、変数が制約条件を満たすよう、与えられた方向に値を修正していきます。(訳注：すなわち制約条件自体を変更するのではなく、引数である入力変数を変更している）従って、このメソッドは `const` にすべきでしたし、メソッド名も別のものにすべきでした。たとえ子供がと駄々をこねるように、開発者が“I can’t”と言ったとしても。実際には、なんとか修正出来たでしょう。

ふたつ目は、このクラスは pimpl idiom（Pointer to Implementation）を使っています（これについては既に説明しています）。デフォールトのコンストラクターは任意の引数として Impl インスタンスに対するポインターを取っています。もし今このクラスを再設計するとしたら、まず引数を取らないデフォールトのコンストラクターを作成します。その上で別途 Impl へのポインターを取るコンストラクターを作成し、それを protected として宣言して派生クスでしか使えないようにしたでしょう。

最後に、これらのクラスが `const` を正しく使ったかどうかについて私の考えを短く説明します。端的に言えば、良くありません。いくつかのメソッドは修正可能ですが、いくつかは無理です。例えば、`minimize( )` メソッドを変えると、過去のバージョンとの互換性が失われるでしょう（このメソッドは純粋仮想関数として宣言されており const かどうかという事はメソッドの属性に含まれています）。また、いくつか他の Optimizer クラスも `minimize( )` メソッドを通して別のメソッドを呼び出し、メソッド間のデータのやり取りを、メンバー変数を使って行っているので、同じく互換性の問題が起こります。

（注：いくつかの Optimizer クラスではメンバー変数を `mutable` で宣言しているものがあります。おそらく、かつてはメソッドが `const` として宣言されていた為でしょう。このセクションを書いている時点では、詳しく調べていませんが）

Version1.0 がリリースされる前に、コードを見直す努力をもう少ししていれば、この問題を避けられたかもしれません。若いプログラマーの方は、これを反面教師として学んで下さい。

11.4.4 Statistics: 統計値

Statistics クラス群は、Monte Carlo シミュレーションで生成されたサンプルを集計する目的で作られました（Chapter VI での説明を思い出して下さい）。このクラスが提供するすべての機能は（Decorator パターンを使って）一連の decorator で実装されており、各 decorator が、階層化さ

れてメソッドを追加するようになっています。階層化が多重になっている理由は、私の推察では、decorator の基本階層になるベースクラスが2つ用意されている為であり、その内のいずれかを選択できるようにする為かと思われます。階層化された設計方法により、ユーザーは、基本階層の共通インターフェースに基づいて、機能を追加していく事が出来ます。

ユーザーが選択できる基本階層のひとつは `IncrementalStatistics` クラスで、そのインターフェースを下記 Listing A.17 に示します。このクラスがどの様なインターフェースを提供しているかは簡単に推察がつくと思いますが、例えば、サンプル数を返したり、加重されたサンプルの場合はその加重合計であったり、その他様々な統計値を返したりします。

Listing A.17: `IncrementalStatistics` クラスのインターフェース

```
class IncrementalStatistics {
  public:
    typedef Real value_type;
    IncrementalStatistics();

    Size samples() const;
    Real weightSum() const;
    Real mean() const;
    Real variance() const;
    Real standardDeviation() const;
    Real errorEstimate() const;
    // skewness, kurtosis, min, max...

    void add(Real value, Real weight = 1.0);
    template <class DataIterator>
    void addSequence(DataIterator begin, DataIterator end);
};
```

サンプルを追加する場合、1つずつ追加する事もできますし、iterator で指定された配列として追加する事もできます。このクラスの特徴は、引数として渡されたデータを保持せず、そのまま統計値を計算する事です。これは、メモリーを節約するのが目的でした。(Library のプロジェクトが始まった)2000年頃は PC の RAM のサイズは 128MB から 256MB であった為、メモリー確保は今よりもずっと大きな関心事でした。このクラスで実際の計算を行うプログラムコードは、かつては独自に開発されたものでしたが、今では Boost の accumulator library を使っています。

ふたつ目の基本階層は、`GeneralStatistics` クラスで、`IncrementalStatistics` クラスと同じ名前を持つインターフェースに加え、いくつかの独自のメソッドが加わっています。このクラスは、渡されたデータを一旦保持し、その値を返す事が出来るので、追加されたメソッドはそうした機能を提供するものです。例えば、データのパーセンタイルを返す事や、データの並び替えが可能です。このクラスは、`expectationValue()` というテンプレートメソッドを提供しており、ユーザーが独自に開発した計算アルゴリズムを使う事が出来ます。興味のある方は、この Section の最後にある“Aside”の欄をご覧下さい。

Listing A.18：`GeneralStatistics` クラス

```
class GeneralStatistics {
  public:
    // ... same as IncrementalStatistics ...

    const std::vector<std::pair<Real,Real> >& data() const;

    template <class Func, class Predicate>
    std::pair<Real,Size>
    expectationValue(const Func& f,
                     const Predicate& inRange) const;

    Real percentile(Real y) const;
    // ... other inspectors ...

    void sort() const;
    void reserve(Size n) const;
};
```

次に、外側（訳注：Decorator パターンにおける Decorate する側）の階層について説明します。最初のクラスは、Expected Shortfall や Value at Risk といった、リスク量計算における統計値を計算できるクラスです。（訳注：Value at Risk は、2σ、99% 信頼区間、99% パーセンタイル、といった、ある一定の“閾値における”損失額。Expected Shortfall は“閾値を超える”損失について、その発生確率をかけた条件付き期待値。）これらの値を計算する際の問題点は、サンプルデータのすべてを必要とすることです。IncrementalStatistics クラスの場合、サンプルデータを保持していないので、正確な計算はあきらめて、近似値を計算する事になります。その方法のひとつは、標本集団がガウス分布と同じモーメントを持つと仮定して、サンプルの平均と分散を使い、解析的に計算する方法です。下記の `GenericGaussianStatistics` クラスは、まさにそれを行っているクラスです。

Listing A.19：`GenericGaussianStatistics` クラスのインターフェース

```
template<class S>
class GenericGaussianStatistics : public S {
  public:
    typedef typename S::value_type value_type;
    GenericGaussianStatistics() {}
    GenericGaussianStatistics(const S& s) : S(s) {}

    Real gaussianDownsideVariance() const;
    Real gaussianDownsideDeviation() const;
    Real gaussianRegret(Real target) const;
    Real gaussianPercentile(Real percentile) const;
    Real gaussianValueAtRisk(Real percentile) const;
```

```
        Real gaussianExpectedShortfall(Real percentile) const;
        // ... other measures ...
    };

    typedef GenericGaussianStatistics<GeneralStatistics>
                                            GaussianStatistics;
```

すでに述べた通り、またコードを見ても判りますが、このクラスはベースクラスに対する Decorator として実装されています。Decorate される対象（ベースクラス）をテンプレート引数として取り、そこから派生するクラスを定義する事によって、ベースクラスのすべてのメソッドが使えた上で、さらに新しいメソッドが追加できます。QuantLib では、テンプレート引数を特定して具体化されたクラス型として、GaussianStatistics クラスを提供していますが、テンプレート引数として GeneralStatistics クラスを指定しています。おかしいですね。テンプレート引数として IncrementalStatistics クラスの方が指定されるべきですが、Library の中に見当たりませんでした。これには理由があります。少しお待ちください。

ベースクラスがサンプルデータをすべて保持しているのであれば、（分布から解析的に計算されるリスク量では無く）データから実際のリスク量を計算する Decorator クラスを作る事ができるはずです。それが、下記 Listing A.20 にある GenericRiskStatistics クラスになります。ここではガウス分布に従った統計量計算アルゴリズムの実装内容については説明しません（各自 Library のコードで確認して下さい）。

Listing A.20：GenericRiskStatistics クラスのインターフェース

```
    template <class S>
    class GenericRiskStatistics : public S {
      public:
        typedef typename S::value_type value_type;

        Real downsideVariance() const;
        Real downsideDeviation() const;
        Real regret(Real target) const;
        Real valueAtRisk(Real percentile) const;
        Real expectedShortfall(Real percentile) const;
        // ... other measures ...
    };

    typedef GenericRiskStatistics<GaussianStatistics>
                                              RiskStatistics;
```

コードを見れば判る通り、Decorator の階層が重ねられています（訳注：コードの最後にある typedef を使ったテンプレートクラスの具体化の部分）。すなわち、ライブラリーでは、GenericRiskStatistics のテンプレート引数に GaussianStatistics クラスを指定して、RiskStatistics という具体的クラスを定義しています（訳注：それ自体が Decorator である

GaussianStatistics クラスを、さらに Decorate するクラスを定義している)。従って、このクラスは、ガウス分布を前提にして解析的に導出した統計値に加え、実際のデータを使って導出した統計値も算出できます。これが、GaussianStatistics の Decorate 対象として GeneralStatistics クラスが選択された理由のひとつでした。

以上に加えて、さらに別の Decorator を重ねる事も可能です。QuantLib ライブラリーの中でも、追加の Decorator がいくつか提供されていますが、ここではプログラムコードは示しません。そのひとつに、SequenceStatistics クラスがあります。このクラスは、標本集団がひとつでは無く、(標本集団の)配列として存在している場合に使われます。このクラスは、内部的にはスカラーの Statistics インスタンスの配列を使っており、各標本集団間の相関や共分散の計算アルゴリズムが追加されています。このクラスは、例えば、LIBOR Market Model のような、異なる LIBOR 期間のキャッシュフローについて、それぞれに標本集団が必要な場合に使われます。その他には、ConvergenceStatistics クラスや DiscrepancyStatistics クラスがあり、これらは、標本集団の配列に関する情報を計算して提供しています。このクラスは Library の他の所では使われていませんが、少なくとも単体テストはやりました。

Aside: 極端な期待

GeneralStatistics クラスを見直してみて、このコードを自慢すべきなのか、恥ずべきなのか、よく判りません。なぜなら、当時私は、このクラスで取り入れた汎用的な設計について、うまくいったと浮かれていました。

それは演算方法の数学的な抽象化でした。当初は(様々な統計値の計算アルゴリズムを)それぞれシンプルに実装するだけでした。しかし、(様々な Decorator の階層で実装された)平均、分散、さらにもっと複雑な計算アルゴリズム(訳注;おそらく3次や4次のモーメントである Kurtosis や Skew の計算の事)を見てみると、すべて同じような形の次のような式で表せると気付きました。

$$\frac{\sum_{x_i \in \mathcal{R}} f(x_i) w_i}{\sum_{x_i \in \mathcal{R}} w_i},$$

すなわち、この式は領域 $\mathcal{R}$ における関数 $f(x)$ の期待値を計算しています。この式を使えば、平均の計算は、$f(x)$ に identity 関数(単に x を返す関数)で領域 $\mathcal{R}$ はサンプル集合全体を指定すればいいでしょう。分散の計算は、$f(x)$ に $(x - \bar{x})^2$ (但し $\bar{x}$ は標本の平均)を使い、同じ領域で計算すれば求まるでしょう。同様に、$f(x)$ の内容を変えるだけでより高次のモーメントも求まります。その結果、expectationValue() メソッドをテンプレートメソッドにし、引数として関数 $f(x)$ と領域 $\mathcal{R}$ を渡し、計算結果とサンプル数を返すようにしました。(平均や分散値を使う)他のメソッドは、適当な引数を渡してこのメソッドを呼び出すように実装されています。例えば、平均値の計算は、次のように実装されており、一見しただけでは何を計算しようとしているのか判りにくいかも知れません。

```
return expectationValue(identity<Real>(),
                        everywhere()).first;
```

ところで、当時、私は functional programming について、もっと学んでおくべきでした。サンプルの有効領域は関数を使って渡されており(訳注;第2引数の everywhere() がそれ)、各サンプルが

領域の有効範囲内かどうかで true または false を返します。上記コード中の everywhere() は、その機能を持った Helper 関数のひとつです。このように（統計値の計算アルゴリズムを抽象化・一般化できた事は）うまくいったと感じていました。誰かが $(x - \bar{x})^2$ を次のように書くまでは。

```
compose(square<Real>(),
        std::bind2nd(std::minus<Real>(), mean()))
```

なるほど、このような書き方は優れて汎用的で、ユーザーが新しい計算方法をプログラムしやすくなります。しかし、やや数学的になりすぎて、意味がわかりにくいプログラムコードでもあります。私の方法の方が、もしかしたら汎用性と解りやすさのバランスがうまく取れているかも知れません。C++11に準拠して上記の bind 関数をラムダ関数に置き換えれば、もう少し解りやすくなるかも知れません。将来、C++11に移行していく際に、どのようにコードが変わっていくか、見てみたいと思います。

一方で、計算速度については、大きな問題になっていません。expectationValue() や compose()、everwhere() メソッドはテンプレートメソッドになっており、コンパイラーが、適切に in-line 化するなど、効率的にコンパイルされるはずです。そうすると、コンパイル後の計算アルゴリズムは、平均や分散の計算アルゴリズムを直接プログラムで書いた場合と同じ様な、より単純なループを使った計算方法になっているはずです。

11.4.5 Linear Algebra: 線形代数（ベクトルと行列）

下記に Array クラスと Matrix クラスの概要を示しますが、これらについて説明すべき事は、それ程ありません。

Listing A.21: Array クラスと Matrix クラスの概要

```
class Array {
  public:
    explicit Array(Size size = 0);
    // ... other constructors ...
    Array(const Array&);
    Array(const Disposable<Array>&);
    Array& operator=(const Array&);
    Array& operator=(const Disposable<Array>&);

    const Array& operator+=(const Array&);
    const Array& operator+=(Real);
    // ... other operators ...

    Real operator[](Size) const;
    Real& operator[](Size);
```

```
    void swap(Array&);
    // ... iterators and other utilities ...
  private:
    boost::scoped_array<Real> data_;
    Size n_;
};

Disposable<Array> operator+(const Array&, const Array&);
Disposable<Array> operator+(const Array&, Real);
// ... other operators and functions ...

class Matrix {
  public:
    Matrix(Size rows, Size columns);
    // ... other constructors, assignment operators etc. ...

    const_row_iterator operator[](Size) const;
    row_iterator operator[](Size);
    Real& operator()(Size i, Size j) const;

    // ... iterators and other utilities ...
};

Disposable<Matrix> operator+(const Matrix&, const Matrix&);
// ... other operators and functions ...
```

このクラスが提供しているインターフェースは、皆さんの予想通りのものです。すなわち、コンストラクターと代入演算子 (=)、配列要素へのアクセス演算子（注：Array クラスは a[i] という記法を使い、Matrix クラスでは、ユーザーにより便利なように、m[i][j] と m(i,j) という2種類の記法を用意しています）、さらに配列や行列の要素毎に行われる算術演算子とユーティリティーメソッドです。
（注：a * b という算術演算子が、ドット積ではなく、要素毎の積を返すように設計されている点は、どうも気になります。でも、それは私だけのようで、他のプログラマー達は気にならないようです）（訳注：ソースコードを見ると、* 演算子は、スカラー積と内積の両方で使えるようオーバーロードされており、内積については、別途 DotProduct( ) というメソッドが Array クラスにだけ用意されている。）

このクラスには、要素サイズを変更するメソッドや、通常のコンテナクラスが持っているようなメソッドはありません。理由は、これらのクラスは数学的演算の為に使われるクラスであり、コンテナクラスとして使われる事を想定していないからです。データの保管は、単純な scoped_ptr を使って行っており（訳注：上記コードでは、boost::scoped_array を使っている）このクラスがメモリーの生成消去管理を行っています。

Array クラスでは、Abs( ) や Log( ) といったメソッドも提供しています。C++ 開発者コミュニティ

ーでの良き開発者でいるために、これらのメソッドを `std` の namespace にある同じ名前のメソッドをオーバーロードしないようにしています。なぜなら、C++ の標準でそれが禁止されているからです。より複雑な計算機能（例えば、行列の平方根や、様々な行列の分解方法など）は、別のモジュールで提供されています。

端的にまとめると、このクラスはベクトルや行列の基本的な機能を忠実に実装しているだけです。一点だけ、意味が明解でない内容の部分は、`Disposable` クラステンプレートを使っている所ですが、この Appendix の別のセクションで詳しく説明します。ここでは、このクラスを使用したのは、C++11の標準が発表される前に、move semantics と同様の効果を試しただけ、と述べておきます。すなわち、抽象化によるメモリー負荷を軽減したかったという事です。演算子のオーバーロードは大変便利なものです。`c=a+b` という記述の方が、`add(a,b,c)` という記述よりもはるかに解りやすいですが、デメリット無しでという訳にはいきません。例えば、

```
Array operator+(const Array& a, const Array& b);
```

という関数宣言では、この演算子は新しい `Array` インスタンスを生成し、それを返すようにしなければなりません。すなわち、その為にメモリーを確保し、そのメモリーをコピーする動作が必要になります。そうすると、演算の回数が増えるほど、それにかかる計算負荷は大きくなっていきます。

ライブラリーの最初のバージョンでは、その問題に対し、expression templates（式テンプレート）を使って対応しようとしていました。（以下にそれを簡単に解説しますが、詳細は巻末(Veldhuizen, 2000) を参照して下さい。）その考え方は、演算子は Array のインスタンスを返すのではなく、式中の各項を示す参照を持つ parse tree のようなものを返します（訳注：parse tree とは、式の構文を解析して演算の順序を示すフローチャートに変換されたもの）。従って、例えば、`2*a+b` といった算術表現は、実際にその計算をするのではなく、計算に必要な情報を持ったアルゴリズムの構造を生成するようにします。式テンプレートに配列が代入された時にだけ、算術式が解析され、その時点でコンパイラーが内容を解析し、計算ループをひとつ生成し、計算を実行して結果を代入された配列にコピーします。

このプログラム技術は今でも有効（コンパイラーの技術が進歩した今では、より重要）ですが、我々は開発開始から数年後に、この方法を止める事にしました。理由は、コードを読むのが難しく、メンテナンスが大変な事です。コードの複雑さについては、今の実装コードと、かつての下記のようなコードを比べてみて下さい。

```
VectorialExpression<
    BinaryVectorialExpression<
        Array::const_iterator, Array::const_iterator, Add> >
operator+(const Array& v1, const Array& v2);
```

それと、すべてのコンパイラーがこの処理を実行できず、expression templates とより単純な実装コードの両方をメンテナンスせざるを得なかったからです。従って、C++ の開発者コミュニティーで move semantics の話が開始され、その実装方法のアイデアが登場した際に、それをヒントにして `Disposable` クラスを使う事にしました。

既に述べた通り、コンパイラーの性能が近年大幅に向上し、すべてのコンパイラーがexpression-template の実装に対応するようになり、その技術も向上しました。しかし、仮に今プログラムコードを設計するとしたら、問題は `Array` や `Matrix` クラスをそもそも設計する必要があったかどうかです。最低限の機能は、`std::valarray` を使って自分で設計し実装できると思います。しかし、boost ライブラリーが提供している uBLAS といった既存のライブラリーを使うのがいいでしょう。これらは、数値計算のコードの専門家によって書かれたものであり、すでに QuantLib ライブラリーの中で、いくつか使われています。

11.5 Global Settings: ライブラリー全体の変数と定数の設定

下記 Listing A.22に示す `Settings` クラスは、Singleton（後で説明します）の一種で QuantLib ライブラリー全体に渡る情報を格納するオブジェクトです。

Listing A.22: `Settings` クラスの概要

```
class Settings : public Singleton<Settings> {
  private:
    class DateProxy : public ObservableValue<Date> {
        DateProxy();
        operator Date() const;
        ...
    };
    ... // more implementation details
  public:
    DateProxy& evaluationDate();
    const DateProxy& evaluationDate() const;
    boost::optional<bool>& includeTodaysCashFlows();
    boost::optional<bool> includeTodaysCashFlows() const;
    ...
};
```

このクラスのメンバー変数は、ほとんどが（何等かの状態を示す）フラッグで、それぞれの意味については 正式な Documentation をご覧下さい。仮にその意味が判らなくても、ライブラリーの使用は問題ありません。ユーザーがきちんと管理しなければならない情報は Evaluation Date（評価日）だけです。そのデフォールト設定は“本日”で、それは（通常）金融商品の時価評価基準日であり、かつ各種インデックスの Fixing 値を決める日でもあります。

この情報をどう取り扱うかは難問でした。金融商品の価格計算は、Evaluation Date に依存しており、日付が変更になれば当然、その通知を受ける必要があります。その機能は、必要な情報を、proxy クラス（訳注：特定のクラスのインスタンスをメンバー変数として取り込み、別の機能を加えた上で、あたかもその特定クラスの代理として使われるクラス）の中に包み込んで、間接的に

返す事によって実現されています。このクラスにある、その機能を担うメソッド群を見れば、それが判ります。この proxy クラス（訳注：上記コードにある DateProxy クラス）は ObservableValue クラステンプレート（次の Listing A.23参照）から派生しています。このクラスは、黙示的に Observable クラスに転換され、さらに代入演算子をオーバーロードして、データ変更時に（登録されている Observer 群に）変更通知を送る機能を与えられています。それと同時に、この proxy クラスは、内包されたデータ（訳注：ここでは Date クラスのインスタンス）に自動的に変換する機能も持っています。

Listing A.23：ObservableValue クラスの概要

```
template <class T>
class ObservableValue {
  public:
    // initialization and assignment
    ObservableValue(const T& t)
    : value(t), observable_(new Observable) {}
    ObservableValue<T>& operator=(const T& t)  {
      value_ = t;
      observable_->notifyObservers();
      return *this;
    }
    // implicit conversions
    operator T() const { return value_; }
    operator boost::shared_ptr<Observable>() const {
      return observable_;
    }
  private:
    T value_;
    boost::shared_ptr<Observable> observable_;
};
```

このクラスを使えば、今説明した機能を、自然な構文で使えます。

まず、Observer インスタンスを Observable（訳注：ここでは evaluationDate() が返す DateProxy インスタンス）に登録するには、次の様な構文で出来ます。

```
registerWith(Settings::instance().evaluationDate());
```

また、次のような構文で、evaluationDate() の返し値を、Date インスタンスのように使えます。

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

この構文は、（evaluationDate() が返す DateProxy インスタンスを）自動的に Date 型に変えています（訳注：オーバーロードされた代入演算子”＝”が、その機能を提供している）。

さらに3点目として、下の構文のように evaluationDate( ) が返す DateProxy インスタンスに、特定の日付を代入できます。（訳注：これも代入演算子“＝”をオーバーロードして機能させている。）

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

さらに、この代入動作の直後、日付の更新があった事を、すべての Observer に通知します。

* * *

もちろん、誰もが気づいている問題は、グローバル変数としての Evaluation Date（時価評価基準日）しか持っていないという点です。そこから来る明らかな弱点は、異なる Evaluation Date を使って商品の時価評価を、並行して行えないという事です（少なくとも、QuantLib ライブラリーの Configuration（環境設定）をそのまま使うとすればですが）。

という事は、環境設定を操作すれば、何とかなりそうですが、問題は簡単ではありません。まず、（ユーザーが若干手を加えれば）Settings インスタンスをスレッド毎に持たせるようにコンパイル時のフラッグを設定する事は可能ですが、後で説明する通り、これですべての問題が解決される訳ではありません。また別の問題として、シングルスレッドでプログラムを走らせていても、この Global 変数はあまり好ましくない現象を起こす可能性があります。仮に、あるひとつの商品だけを、異なる Evaluation Date を使って時価評価した場合、Evaluation Date を元の日に戻した時点で、その日付変更によって、他のすべての商品の時価の再計算を起動してしまうことになります。

この状況を解決するには、Global Settings クラスを何等かの context クラス（訳注：どのスレッドで、そのプロセスが選択されたのかを示す指標を管理するクラス）に取り換えるべきとの方向性を示唆しています（この問題について、我々の間で話をした時、何人かの鋭い人からも同じアイデアが出ました）。しかし、ある特定の価格計算において、どうやって context を選択すればいいのでしょうか？

Instruments クラスの中に `setContext()` メソッドを加えるアイデアは、使えそうに見えるかも知れません。そうする事によって、計算の際に、Instruments は価格エンジンに対し context を伝え、その価格エンジンからさらに価格計算に使われる TermStructure インスタンスなどにそれを伝達する方法です。しかし、このロジックを実装するのは簡単ではありません。

まず、Instruments インスタンスも価格エンジンインスタンスも、価格計算に必要な TermStructure クラスが何か常に判っている訳ではありません。例えば、Swap は、いくつかのクーポンキャッシュフローを持っていますが、その一部は価格計算において Forecasting カーブの情報を参照している可能性があります。その情報を得る為には、関連するすべてのクラスに、必要な機能を付け加える必要があります。そもそも Coupon オブジェクトに context オブジェクトを設定するのは、いいアイデアとは思いません。

次に、そしてより重要な点ですが、価格エンジンに context を設定する動作は、データが変更される動作 (mutating operation) を内包しています。Instruments に、その context 設定の役割をまかせると、（Instruments から）`const` メソッドであるはずの `NPV()` メソッドを呼び出している最中に、価格エンジン中の context 情報が変更されてしまう可能性があります。この状況は、簡単に race condition（同じ動作で異なった計算結果を出す状況）を発生させます。例えば、一見害のなさそうな計算として、2つの商品が、異なる Evaluation Date を基準に、同じ Discount カーブを使って時価評価しようとすると、どうなるでしょう。並列処理プログラミングの経験がほとんどないユー

ザーが、夢にも思わなかった事が起こり得ます。例えば、同時に走っている2つのスレッドから、同じ（TermStructure インスタンスへの）ハンドルにリンクされてしまう事があり得ます。仮にそうなっても、データ変更の動作が `const` メソッドの中に隠されているので、ユーザーはそれに気づかない可能性があります。

（注：“ちょっと待って。`NPV()` を呼び出している間に、他にもデータを変更するような動作があるのでは？”と言われるかも知れません。よく気づかれました。このセクションの最後にある Aside の欄を見てください。）

そうすると、どうも価格計算を始める前に context を設定する必要がありそうです。であれば、Instruments にそれをまかせる方法は排除されます（なぜなら、（価格エンジンに context を設定する場合と）同様に、ある商品に context を設定した場合、その商品が使う TermStructure インスタンスについて、同じインスタンスを既に使っていた別の商品により設定された context を無効にしてしまう可能性があるからです）。おそらく、価格計算に使う複数の TermStructure インスタンスに、明示的に context を設定しなければならないでしょう。その方向性を取った場合のメリットは、知らずに同じオブジェクトに対し同じ context を設定しようとして race condition に陥るリスクを無くせる事です。逆にデメリットは、設定がより複雑になる事と、異なる context を同時に設定する場合は、TermStructure インスタンスの複製を作る必要があるという事でしょう。例えば異なる Evaluation Date での価格計算を並行処理する場合、OIS ディスカウンティングカーブを2つ用意しないといけなくなります。かつ、その場合、スレッド事の Singleton を管理しないといけなくなるでしょう。

最後に、context が渡されても保存されないような場合について簡単に触れましょう。その場合、メソッドの呼び出し方法が次のような形になるでしょう。

```
termStructure->discount(t, context);
```

これだと、cashing を完全に壊し、すべての関係者に不愉快な思いをさせそうです。もしこのような物を望むなら、Haskell でプログラムを書いたでしょう。

まとめると、このセクションンを暗い気持ちで終わるのは嫌ですが、すべてがうまく行く方法はありません。Global 変数の設定は制約になりますが、私自身解決策を持ち合わせていません。さらに悪いことに、取り得る変更方法は、複雑さを増大させます。QuantLib ライブラリーを初めて使う人に、Black-Scholes の公式を探しているのなら、TermStructure や Quotes や Instruments や Engine といったクラスだけでなく、context も必要ですと説明する事になるでしょう。このセクションの説明で少しは役に立ったでしょうか？

Aside: B 級映画以上に mutation が登場

残念ながら、`const` メソッドであるはずの `Instrument::NPV()` が呼び出されている最中に、データが変更されてしまう事態は、他にもありました。

まず、価格エンジンの中に取り込まれる `arguments` と `results` インスタンスから始めましょう。これらのインスタンスは価格計算の最中に読み込まれ、そして書き込まれます。従って、同じ価格エンジンのインスタンスを使って、他の Instruments の価格計算を同時に行う事はできません。この点は、価格エンジンに（鍵）を加える事で修正できるかも知れません（そうすると、複数商品の価格

計算を順番にですが連続して行えます)、あるいはインターフェースを変更して、価格エンジンクラスの `calculate()` メソッドが `arguments` 構造体を引数として取り込み、`results` 構造体を返すようにしても対応できるでしょう。

次に、Instrument クラスそのものの中に `mutable` なメンバー変数が含まれており、それらは価格計算の最後に書き込まれます。これが問題になるかどうかは、実行しようとしている計算の種類によります。その Instrument の価格を2回、同時に走っている2つのスレッドで計算すると、同じメンバー変数に計算結果が2回書きこまれて終わりになる可能性があります。

最後に思い浮かぶのは、hidden mutation(隠れたデータ変更)で、おそらくそれが最も危険でしょう。TermStructure インスタンスを価格計算の最中に使おうとすると、Bootstrap の処理を起動する可能性があり、別々のスレッドで行われている価格計算が同じ TermStructure インスタンスを使って同時に走ってしまうと、お互いの計算結果をおかしくしてしまう可能性があります。Bootstrapping の計算ロジックが再帰的であるので、(ひとつの計算プロセスの最中に他のエンジンからのアクセスを制限する為の)鍵をかける事が可能かどうか、確信がありません。従って、もしユーザーの方が、(注意深く、すべての設定をあらかじめ行い、同じ Evaluation Date を使う条件で)並列計算を実行しようとするなら、計算をスタートする前に、Bootstrapping のプロセスを完全に終わらせてから計算するようにして下さい。

11.6 Utilities: ユーティリティークラス

QuantLib では、金融に関する概念以外の概念も、いくつかのクラスや関数によってオブジェクトモデル化されています。これらは、ライブラリー全体の土台を築くのに使われるボルトやナットのような役割をしています。その内のいくつかについて、このセクションで説明します。

11.6.1 Smart Pointers and Handles: スマートポインターとハンドル

実行時 polymorphism (多相性)を使っている事により、(大半とは言いませんが)多くのオブジェクトが Heap メモリー上に領域を割り当てられています。この事は、メモリー管理の問題を発生させます。他のプログラム言語では、その役割は備え付けの Garbage Collection (Heap 上の不要なメモリー領域を管理し消去する機能)が行っていますが、C++ では開発者自身で管理する必要があります。

メモリー管理は、殆どが過去の問題で、現時点で多くの問題が残っている訳ではありません。この問題への対応はやっかいなので(特に、例外処理が発生した場合)、マニュアル管理ではなく、できるだけ自動的にできるような方法を探してきました。

C++ 開発者にとって、この問題と戦う為の最善の武器は、スマートポインターです。これは、C++ に備え付けのポインターと同じ機能を持つと同時に、ポインターが指し示すオブジェクト用に、メモリーの生成や維持管理を行い、また不要になれば破棄するという動作まで行っています。このような機能を持つクラスの実装例はたくさんありますが、我々は Boost ライブラリーが提供しているスマートポインターを選択しました(特に `shared_ptr` で、今では ANSI/ISO の C++ 標準ラ

イブラリーにも含まれています)。このポインターの詳しい説明は Boost ライブラリーの文書を見て下さい。QuantLib は、このスマートポインターを使う事により、メモリー管理が完全に自動化されているという点だけ述べておきます。様々なオブジェクトがあちこちで、動的にメモリー配分されていますが、QuantLib の何万行ものコードの中で、ひとつも `delete` を使っていません。

“ポインターへのポインター”も、スマートポインターと同じ様な機能(訳注:メモリーの領域取得と破棄を自動的に行う機能)をするクラスを使って用意しています(“ポインターへのポインター”について再確認したい方は、このセクションの最後の Aside をご覧下さい)。我々は、単にスマートポインターへのスマートポインターを使うような選択をしませんでした。仮にそうすると、次のようなコードになりますが、Emacs (訳注:UNIX 環境のプログラマーに人気のテキストエディタで、キーボード操作だけで素早くプログラムを書ける)を使ったとしても、長たらしいコードを書くのは面倒です。

```
boost::shared_ptr<boost::shared_ptr<YieldTermStructure> >
```

また、この構文の中の内側のポインターは、動的に配分される必要がありますが、あまり気持ちよくありません。さらに、(訳注:Observable への監視が、さらに間接的になるので)Observer パターンを実装するのが難しくなります。

そこで、QuantLib では、`Handle` というクラステンプレートを用意しています。その実装内容を下記 Listing A.24 に示しますが、このクラスの機能は、スマートポインター(`shared_ptr<Type>`)をメンバー変数に持つ“Link”という介在役の内部クラスを利用しています。Handle クラスは、それを使って、Link 型インスタンスへのポインター(`shared_ptr<Link<Type>>`)をメンバー変数として保持し、かつそれを使い易くする為のメソッド群を用意しています。ある Handle インスタンスのコピーは、すべて同じ Link インスタンスを共有するので、そのリンク先のオブジェクトに変更があった場合でも、それがすべての Handle インスタンスに自動的に反映されます。

Listing A.24:`Handle` クラステンプレートの概要

```
template <class Type>
class Handle {
  protected:
    class Link : public Observable, public Observer {
      public:
        explicit Link(const shared_ptr<Type>& h =
                                      shared_ptr<Type>());
        void linkTo(const shared_ptr<Type>&);
        bool empty() const;
        void update() { notifyObservers(); }
      private:
        shared_ptr<Type> h_;
    };
    boost::shared_ptr<Link<Type> > link_;
  public:
    explicit Handle(const shared_ptr<Type>& h =
```

```
                                        shared_ptr<Type>());
        const shared_ptr<Type>& operator->() const;
        const shared_ptr<Type>& operator*() const;
        bool empty() const;
        operator boost::shared_ptr<Observable>() const;
    };

    template <class Type>
    class RelinkableHandle : public Handle<Type> {
      public:
        explicit RelinkableHandle(const shared_ptr<Type>& h =
                                        shared_ptr<Type>());
        void linkTo(const boost::shared_ptr<Type>&);
    };
```

さらに Handle クラスは、`shared_ptr<Link<Type>>` 型の変数によって、他のクラスの Observable となる機能も与えられています。即ち、Link クラスは Observer と Observable から派生しており、自分が監視している Observable から（データ更新の）通知を受けた場合、それを、自分を監視している Observer に転送する機能を持ちます。また、自分がポイントしているオブジェクトが入れ替わった場合にも、同様に Observer に通知します。Handle クラスは、この機能を、Link へのポインターを返す `shared_ptr<Observable>` オペレーターの自動型変換を定義する事によって実現しています。従って、次のような宣言は構文上許されており、期待通りの動作をします。

```
registerWith(h);
```

登録された Observer は、Link インスタンスと（間接的に）それが指し示すオブジェクトから通知を受ける事になります。

Handle を別のオブジェクトに再リンクする（すなわち、ある Handle インスタンスのコピーも同時にすべて、リンクし直した後の別のオブジェクトを指し示す）方法が、Handle クラスそのものではなく、派生クラスである `RelinkableHandle` クラスに与えられている点に気づかれたでしょうか。このようにした理由は、再リンクの機能を使う事ができる Handle インスタンスと、その機能を使えない Handle インスタンスを、区別して間違えないようにする為です。Handle クラスの典型的な使用例では、Handle インスタンスが生成された後、様々な金融商品オブジェクトや価格エンジン、その他のオブジェクトに渡されて、それらのオブジェクト毎に、その Handle インスタンスのコピーが保存され、使用されます。ここで重要な点は、Handle のコピーを保持しているオブジェクトに Handle が指し示しているポインターの再リンクを、どのような理由があっても、許してはならないという事です（あるいは、そのオブジェクトのインスペクター関数を使って、オブジェクトの外から Handle インスタンスにアクセスできるようにしている場合には、その外からのアクセス元にも同様）。仮にそうしてしまうと、他のオブジェクトにひどい影響を与える可能性があります。（注：そういった影響は、思ったよりも頻繁に起こり得ます。実際に、それでひどい目にあいました）

再リンクは、オリジナルの Handle インスタンス（Master Handle とでも呼びましょうか）からのみ、変更が許されるべきなのです。

人間にその管理を任せると間違いを起こしやすいので、我々は、それをコンパイラーに強制させるようにしました。しかし、`linkTo()` メソッドを `const` にした上で、インスペクター関数から、`const` な Handle を返すような方法では、機能しません。なぜならクライアントコード（ユーザーが独自に開発したコード）で Handle インスタンスの non-`const` なコピーが、簡単に作れてしまうからです。そこで、我々は、`linkTo()` メソッドを Handle クラスのインターフェースから取り除き、派生クラスに移しました（訳注：再リンクの機能を、`RelinkableHandle` クラスという特別な派生クラスにのみ許可している）。C++ にある型変換の仕組みを使えば、これでうまくいきます。まず、マスター Handle の役割を持つインスタンスを、RelinkableHandle クラスを使って生成します。そのインスタンスを、同じポイント先のデータを必要とする他のオブジェクトに渡します。その際、派生クラスからベースクラスへの自動型変換が働き、そのオブジェクトは完全に機能する Handle インスタンスのコピーを受け取ります。一方で、Handle インスタンスのコピーがインスペクター関数から返された場合、それを RelinkableHandle へ Down-Cast する方法はありません。（訳注：その結果、Link 先のポインターの変更は RelinkableHandle のマスターインスタンスでしか行えないことになる）

Aside: ポインターに関する C++ 構文上の決まり

あるクラスのインスタンスがポインターのコピーを保持する場合、そのインスタンスはポインターが指し示す先の現在の値にアクセスする事ができます。次のコードを見て下さい。

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

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

しかし、インスタンスの変数に代入されたポインターは、もともと当初のポインターのコピーになりますが、その当初のポインターがインスタンスの外で変更されたとしても、クラスインスタンスに保持されているポインターはその影響を受けません。

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

通常、これの解決策は、間接的アクセスをもう一段階加える事です。Foo クラスを修正して、ポインターのポインターを格納するようにすると、以下のように、上記2種類の変更の両方に対応可能です。

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

11.6.2 Error Reporting: 例外処理

QuantLib ライブラリーの中には、入力値の条件チェックを行うべき場所が至る所にあります。このチェックの方法については、

```
if (i >= v.size())
    throw Error("index out of range");
```

という風にやるのでは無く、次の構文のようにチェックの意図がより明確に解るような方法を取りたいと考えました。

```
require(i < v.size(), "index out of range");
```

この構文であれば、条件が満たされない事をチェックするのではなく条件が満たされる事を要求するような表現になります。さらにそれ自体の意味が自明な require や ensure や assert という単語を使って、チェックの意図（入力値のチェックなのか、出力値のチェックなのか、プログラムのエラーチェックなのか）が解るようなプログラムコードが書けます。

我々は、そのような syntax（構文）を、マクロを使って提供しています。読者の方が“そんなの止めてくれ”と言っているのが聞こえます。その通り、マクロは良くないと思われているし、実際にこれによって若干の問題も発生しています。それについては下記で説明します。しかしこの場合は、関数を使う方がより大きなデメリットをもたらします。なぜなら関数は、すべての引数を評価するからです。次のような、やや複雑な例外処理のメッセージを生成する場合が多々あります。

```
require(i < v.size(),
    "index " + to_string(i) + " out of range");
```

この場合、もし require() が関数であるなら、(2つめの引数である)メッセージの部分は、条件が満たされようが無かろうが、必ず生成されます。そうすると、場合によっては許容できないほどのパフォーマンス悪化をもたらします。もし同じ構文をマクロで作ると、マクロの部分は実際には次のような構文で置き換えられます。

```
if ( !( i < v.size() ) )
    throw Error ("index " + to_string(i) + " out of range");
```

こうすれば、例外処理のメッセージは、条件が満たされなかった場合のみ生成されます。

次の Listing A.25 は、そういったマクロのひとつ QL_REQUIRE の現バージョンを示しています。その他のマクロも同じ様に定義されています。

Listing A.25:QL_REQUIRE マクロの定義

```
#define QL_REQUIRE(condition,message) ¥
if (!(condition)) { ¥
    std::ostringstream _ql_msg_stream; ¥
    _ql_msg_stream << message; ¥
    throw QuantLib::Error(__FILE__,__LINE__, ¥
                          BOOST_CURRENT_FUNCTION, ¥
                          _ql_msg_stream.str()); ¥
} else
```

この定義の方法を使えば、エラー時に、さらにいくつかの警告メッセージが出されます。ひとつ目は、メッセージを生成するのに標準ライブラリーの ostringstream を使っているので、次のような構文が使えます。

```
QL_REQUIRE(i < v.size(),
           "index " << i << " out of range");
```

(この方法がどのように動作するか理解する為に、マクロの本体を一部修正して動作確認してみて下さい。) ふたつ目は、例外処理に飛ぶ場合、Error インスタンスに、今走っている関数の名前と例外処理に飛んだコードの line とファイル名が渡されます。コンパイル時のフラッグ設定によりますが、この情報もエラーメッセージの中に含める事が可能で、プログラマーの問題発見の参考になります。QuantLib ライブラリーでのデフォールトの設定は、ユーザーにとっての便益があまり無いので、この情報をメッセージに含めないようになっています。最後に、マクロ構文の最後になぜ else が付いているか疑問に思うでしょう。マクロに共通する落とし穴で、lexical scope の欠如と呼ばれています。else 以下は、次のように何らかのプログラムコードを追加する必要があります。

```
if (someCondition())
    QL_REQUIRE(i < v.size(), "index out of bounds");
else
    doSomethingElse();
```

マクロの中に else が含まれていなかったとすると、上記のコードは意図した通りに動作しません。else をマクロの外でプログラムコードの中に入れた時、マクロ中の if 文に対する else と組み合わされ、次のようなコードが書かれたと解釈されてしまいます。

```
if (someCondition()) {
    if (!(i < v.size()))
        throw Error("index out of bounds");
    else
        doSomethingElse();
}
```

これは、意図した動作ではありません。

最後の注意点として、これらのマクロの不利益な点についても説明しなければなりません。現状のマクロの記述では、例外処理に飛ぶと、引数に含まれているメッセージしか返す事が出来ません。その他のチェックすべきデータのインスペクター関数が全く定義されていません。例えば、インデックスが“範囲外”とのメッセージの中にそのインデックスを含める事が出来ますが、例外処理のプロセスの中にそのインデックスを integer として返すメソッドがありません。従って、その情報はユーザーが Display で見る事は出来ますが、例外処理後にエラーをリカバリーするプログラムコードの中で使う事が出来ません。だれかがメッセージからその情報を切り取る以外は。しかし、そのメリットはそのコードを書く手間に見合いません。従って、今の所、解決策はありません。もし読者の中で良いアイデアをお持ちなら、そのコードを送って下さい。

11.6.3 Disposable Objects (move semantics　の代用)

Disposable クラステンプレートは、C++98 準拠のコードで、move semantics を実現する為の試みでした。(訳注：move semantics については”右辺値参照・ムーブセマンティクス”)　そのアイデアを提唱した人の功績を称える為、そのアイデアを取ってきた原典を紹介します。それは Andrei Alexandrescu 氏の論文で(巻末(Alexandrescu, 2003) 参照)、そこでは、関数が値を返す際に一時的に生成されるオブジェクトのコピーを省略する方法を提示しました。

基本的なアイデアは、彼の文献が発表された当時、すでに広まりつつあったものですが、最終的に C++11で正式に取り入れられました。すなわち、一時的に生成されたオブジェクト(右辺値)を、他のオブジェクト(左辺値)にコピーする場合、場合によっては、その一時オブジェクトを代入先と交換する方が、プロセス効率が上がる場合があります。例えば、一時的に生成された配列変数を動かしたい時(要素をすべて別の配列変数にすべて移す場合)どうしますか？　新しい配列変数を作って、一時的な配列変数が示す Pointee(配列の中身)の先頭アドレスをコピーする方が、配列変数の要素をすべてコピーするより早いですよね。新しい C++ では、rvalue reference(右

辺値参照)の概念を使って move semantics をサポートしています。(巻末(Hinnant *et al*, 2006))

コンパイラーは、一時的に生成されるオブジェクトの生成消去のタイミングが分っています。それを利用すれば(C++11準拠の)`std::move` を使って、一時的に生成されたオブジェクトを、別のオブジェクトに移す事ができます。ところが、我々の実装方法は、次の Listing A.26 にあるように、そのようなコンパイラーのサポートを使わずに実現しようとするものです。その結果どうなったか、すぐ後に説明します。

Listing A.26 `:Disposable` クラステンプレートの実装

```
template <class T>
class Disposable : public T {
  public:
    Disposable(T& t) {
        this->swap(t);
    }
    Disposable(const Disposable<T>& t) : T() {
        this->swap(const_cast<Disposable<T>&>(t));
    }
    Disposable<T>& operator=(const Disposable<T>& t) {
        this->swap(const_cast<Disposable<T>&>(t));
        return *this;
    }
};
```

このクラスそのものに、あまり注目すべき所はありません。このクラスの動作は、`swap()` メソッドを実装する際に使われるテンプレート引数に依存します。このメソッドを使って、引数のインスタンスの中に含まれるデータを(望むべくは効率よく)コピーではなく、スワップします。コンストラクターと代入演算子はすべて、この `swap()` メソッドを使って、インスタンスの中のデータを、コピーせずに新しいオブジェクトに移動させます。

Disposable インスタンスを別の Disposable インスタンスを使って構築する場合は、`const` 参照を使って、構築元のオブジェクトを受取ります。なぜなら引数のオブジェクトを、一時変数のままにしておく必要があるからです(実際に Disposable オブジェクトの殆どは、一時変数です)。従って、実際に `swap()` メソッドを起動した時、Disposable の中身を使う必要があるので、そこで `const_cast` を使う必要があります。(訳注:`const_cast` により引数の Disposable の `const` を外し、内容を変更可能にする為)　Disposable を Disposable 以外のオブジェクトから作る場合は、反対にそれを non-const な参照として取ります。これは、破壊的な型変換を起こしたり、使えると思ったオブジェクトが空っぽだった状況に陥るのを防ぐ為です。この事はしかし、デメリットもあります。それについては、少し後で触れます。

次の Listing A.27 は `Disposable` を(テンプレート引数で特定された)クラスに改造する方法を示しています。ここでは `Array` クラスを例に使っています。

Listing A.27 : `Array` クラスを使った、`Disposable` クラステンプレートの使用法

```
Array::Array(const Disposable<Array>& from)
: data_((Real*)(0)), n_(0) {
    swap(const_cast<Disposable<Array>&>(from));
}

Array& Array::operator=(const Disposable<Array>& from) {
    swap(const_cast<Disposable<Array>&>(from));
    return *this;
}

void Array::swap(Array& from) {
    data_.swap(from.data_);
    std::swap(n_,from.n_);
}
```

ご覧の通り、`Disposable` を引数で取る、コンストラクターおよび代入演算子を追加で定義する必要があります。(C++11では、これらは、rvalue reference(右辺値参照)を取る move コンストラクターと move 代入演算子になります。)それと同時に、コンストラクターと代入演算子の中で使われる `swap()` メソッドも実装しなければなりません。ここでもコンストラクターは、一時変数を拘束する為に、引数を `const` 参照として取り、後で `const_cast` して `const` を外しています。しかし、今思えば、引数をコピーで取るような、もうひとつの効率的な `swap()` メソッドを追加してもよかったかも知れません。

最後に、関数からの返される値として `Disposable` を使う方法についてです。以下のコードをご覧下さい。

```
Disposable<Array> ones(Size n) {
    Array result(n, 1.0);
    return result;
}

Array a = ones(10);
```

`ones()` メソッドが Array 型を返すと、それが `Disposable < Array >`に型変換され、それをさらに Array 型の変数 a に代入すると、その内容が a とスワップされます。ここで、先ほど Disposable コンストラクターが安全に non-const の参照を取る場合のデメリットについて述べて事を思い出して下さい。デメリットとは、一時変数を bind 出来ないという事です。従って、上のコードの `ones()` を以下のように簡略化する事はできません。(訳注:上のコードでは、関数内で一旦 Array 型のローカル変数 result(左辺値)を生成して、それを返すようにしているが、下記コードでは、Array のコンストラクターの返す値(Array 型のインスタンスで右辺値)を関数の返値としている)

```
Disposable<Array> ones(Size n) {
    return Array(n, 1.0);
}
```

なぜなら、このコードはコンパイルできません。その結果、より冗長な構文を強いられ、かつそのArray 配列に変数名を割り当てなければなりません。
（注：実際には強いられている訳ではありません。例えば　上のコードで、`Array` を返すのではなく、`Disposable<Array> (Array(n,10))` を返すようにすれば、ローカル変数をわざわざ生成する必要はありません。しかし、これではローカル変数を生成するよりも、さらに判りにくいでしょう。）

現時点では（C++11以降に準拠すれば）、もちろん右辺値参照と move コンストラクターを使って、以上のような事（訳注：わざわざ `Disposable` クラステンプレートを作り、さらに各クラスでコンストラクターと代入演算子をオーバーロードする作業）を忘れ去る事ができます。さらに正直に申し上げると、`Disposable` はコンパイラーの邪魔をして、意図したメリットよりもデメリットの方が多いのではと疑っています。実は、最新の C++ に準拠すれば、上記のようなシンプルなコードで、かつ抽象化のペナルティーを避ける方法があるのをご存知ですか？　それは、次の様なものです。

```
Array ones(Size n) {
    return Array(n, 1.0);
}

Array a = ones(10);
```

C++17では、Array を返す際や、代入する際に行われていた（一次変数への）コピーの動作が、実は省略されています（すなわち、コンパイラーによって、代入先の変数のメモリー領域を、代入元の領域にあらかじめ確保するようなコードが生成されている）。最新のコンパイラーは、この動作が C++ 準拠と決まる前に、既にこういった省力化を行っていました。コンパイラーが自動的に行ってくれるこの動作は、RVO（Return Value Optimization）と呼ばれています。しかし乍ら、`Disposable` を使うと、コンパイラーが行うこの自動的な動作を妨げ、返って処理速度を遅くしてしまう可能性があります。

11.7 Designe Patterns: デザインパターン

QuantLib ライブラリーの中では、様々なデザインパターンが使われています。デザインパターンの詳細については、有名な Gang of Four（“4人組”）の本（巻末 (Gamma *et al*, 1995) 参照）を参考にして下さい。ではなぜ、ここで私自身がデザインパターンについて書く必要があるのでしょうか？　かつて G.K.Chesterton が記していた次の言葉があります。

> Poets have been mysteriously silent on the subject of cheese（詩人は、“チーズ”については謎のごとく題材にしないものだ）

そして、有名な4人組も、デザインパターンを実装する際に直面する現実的な問題については、何も述べていません（但し、それは彼らの問題では無い点注意して下さい）。デザインパターンの応用例はほとんど無限にありますが、彼らは4人しかいないのです。

そこで、Appendix の最後のセクションを使って、QuantLib ライブラリーの要件を満たす為、我々が使ったデザインパターンの実装方法を説明したいと思います。

11.7.1 The Observer Pattern: オブザーバーパターン

QuantLib ライブラリーでは、至る所で Observer パターンが使われています。既に、Chapter II と III で、Instruments（金融商品）や TermStructure（期間構造）クラスが、このパターンを使って市場データの変動を常にウォッチし、必要な時に価格の再計算を行うのを見てきました。

我々の実装方法は（概要を次の Listing A.28 に示します）4人組の本で説明された方法とほぼ同じです。しかし、先ほど述べた通り、この本では教えてくれていない、いくつかの疑問点や問題点があります。

Listing A.28：`Observer` と `Observable` クラスの概要

```
class Observable {
    friend class Observer;
  public:
    void notifyObservers() {
        for (iterator i=observers_.begin();
                      i!=observers_.end(); ++i) {
            try {
                (*i)->update();
            } catch (std::exception& e) {
                // store information for later
            }
        }
    }
  private:
    void registerObserver(Observer* o) {
        observers_.insert(o);
    }
    void unregisterObserver(Observer*);
    list<Observer*> observers_;
};

class Observer {
  public:
    virtual ~Observer() {
        for (iterator i=observables_.begin();
                     i!=observables_.end(); ++i)
```

```
            (*i)->unregisterObserver(this);
        }
        void registerWith(const shared_ptr<Observable>& o) {
            o->registerObserver(this);
            observables_.insert(o);
        }
        void unregisterWith(const shared_ptr<Observable>&);
        virtual void update() = 0;
      private:
        list<shared_ptr<Observable> > observables_;
    };
```

例えば、(Observable から Observer に送るデータ更新の)“通知”の中に、どのような情報を含めたらいいのでしょうか？　我々は、それを最小限の情報に留めました。即ち、Observer オブジェクトが知るべき情報は、データの更新があったという事だけにしました。もっと多くの情報を渡す事も可能でした(例えば、update( ) メソッドの引数に“通知”を発信している Observable インスタンスを持たせるとか)。そうすれば、Observer 側は何を再計算すればいいのか分るので、CPU クロックの数サイクル分、計算時間を節約できたかも知れません。しかし、それによるメリットは、その結果生じる複雑さのデメリットを上回るとは思いません。

別の疑問点として、Observer の `update()` メソッドの最中に例外処理が発生した場合は、どうなるのでしょう?そのような事態は、ある Observable から“通知”を発信している最中、即ち Observable が登録している複数の Observer インスタンスに“通知”を順次発信している際に、その中のひとつの Observer インスタンスで起こり得ます。もし、そこで例外処理ルーチンに飛んでしまうと、Iterator を使ったループ処理が中断し、残りの Observer インスタンスは“通知”を受け取れません。それは良くないですね。我々が取った方法は、何等かの例外をキャッチした場合でもループ処理は継続して行い、ループ終了後に例外処理ルーチンに飛ぶようにしました。この方法だと、発生した例外に関する情報が失われる難点がありますが、通知が中断する事と比べれば問題は小さいと考えました。

3番目の疑問点に移ります。それはコピーされた際の動作に関するものです。Observer あるいは Observable インスタンスがコピーされた際に、どのように動作するのが正解なのか、あまり明確ではありません。現状は、我々が妥当だと考える方法で実装しています。まず Observable がコピーされた場合、そこに登録されている Observer インスタンスのリストはコピーされないようにしています。一方で、Observer がコピーされた場合、コピー元が登録されていた Observable に、そのコピーも登録されるようにしています。別の方法も選択可能でしたが、実際の所、正しい選択肢は、コピーそのものを禁止する事だったかもしれません。

しかし乍ら、コピーを禁止せず上記の対応をした結果、2つの大きな問題が発生しています。ひとつ目は、Observer と Observable インスタンスの生成・消去を適切に管理しなければなりません。即ち、消去済みの Observer に“通知”が発信されないようにしなければなりません。それを実現する為、まず Observer に Observable インスタンスへの shared pointer を持たせるようにしました。それにより Observable を消去しようとしても、そこに登録されているすべての Observer において Observable の shared pointer を消去するまでは、それが出来ないようにしました。一方で Observer

インスタンスを消去する場合は、それが登録されているすべての Observable で、登録をはずすようにしています。そうすれば、Observable が Observer の生のポインターのリストを保持しても安全です。

ところが、この shared pointer を使った対応については、(そのままでは)シングルスレッドの場合でしかきちんと動作しません。我々は QuantLib の bindings を C# や Java に export しようとしていますが、そこでは残念ながら、別のスレッドで常に Garbage Collection のコードが走って、不要になった Heap メモリー上のオブジェクトを消去しています。その為、Observable からの"通知"が、半分消去された(訳注:文脈から、同じ変数を指す複数の shared pointer の中の一部のポインターだけを消去した状態と想定される)Observer インスタンスに送られると、不規則なクラッシュが起こります。この問題が発見された後、この Bug は boost::shared_ptr に備わっている裏技を使って修正されましたが(Klaus さん、ありがとう)、その裏技については十分な文書が無く、将来これが std::shared_ptr に移行した後も有効かどうか、よく分かりません。また、この裏技により動作スピードが遅くなっています。その為、デフォールトの設定ではこの裏技を使わないようにしており、コンパイル時のスイッチ操作で使用可能に出来る様にしています。従って、もし C# や Java で QuantLib を使う場合にだけ、使用可能にして下さい。

ふたつ目の大きな問題は(この問題は今でも続いています)、まるで Jerry Lee Lewis の歌"Whole Lotta Shakin' Goin' On"のように、大量の"通知"が飛び回る事です(Whole lotta notifin' goin' on)。ひとつのデータ更新が、何十あるいは何百もの通知発信を起動することがあります。ほとんどの update( ) メソッドが、単にフラッグをセットしたり、通知を転送するだけであったとしても、数が膨らめば、時間もそれだけかかります。

QuantLib ライブラリーを実際に、CVA/XVA のような、計算時間が相当かかるようなアプリケーションの中で使っている人の中には、"通知"が自動的に発信される仕組みを止めて、再計算を指示された場合にだけ明示的に行うようにする事で回避しているケースもあります(Peter さん、そうですね?)。"通知"にかかる時間を減少させる為に、通知連鎖の途中にある転送するだけのオブジェクトは飛ばしてしまう方法はうまくいきません。理由は、"通知"の連絡網の至る所で使われている Handle クラスの為です。Handle オブジェクトは re-link が可能で、オブジェクトが生成された後でも連絡網のネットワークが変更されてしまいます。

端的に言えば、この問題は未解決です。もし読者の方が良い解決方法をお持ちなら、連絡先はご存知ですよね。

11.7.2 The Singleton Pattern: シングルトンパターン

かの"4人組"の人たちは、彼らの本の最初の部分を creational パターン(訳注:オブジェクトの生成に関するデザインパターン。Singleton Pattern の他、Abstract Factory Pattern、Builder Pattern、Factory Method Pattern、Prototype Pattern が該当。詳細は Wiki "Creational Pattern" 参照)に費やしました。論理的にはその順番になるのでしょうが、この選択は残念な副作用を生み出す事になりました。この本を読んだ熱心なプログラマー達が、それに倣って、コードのあちこちに Abstract Factory や Singleton クラスのオブジェクトをまき散らしているのを、頻繁に見かけます。言わずもがなですが、その結果、プログラムコードの明瞭さの点で意図せざる事が起こりました。

読者の方は、QuantLib ライブラリーの中で Singleton クラステンプレートがあるのを見て、同じ理由によるものと疑われたかも知れません。(意地悪く言いますけど、それは見当違いです。)幸

いにも、そうで無いという根拠は、バージョンコントロールの記録で示す事が出来ます。このクラスは、QuantLib ライブラリーの中では、Observer パターン（behavioral pattern）や Composite パターン（structural pattern）よりも後に加えられました。

QuantLib ライブラリーにおける Singleton のデフォールトの実装内容は下記 Listing A.29で示す通りです。

Listing A.29：`Singleton` クラステンプレートの実装内容

```
template <class T>
class Singleton : private noncopyable {
  public:
    static T& instance();
  protected:
    Singleton();
};

#if defined(QL_ENABLE_SESSIONS)
// the definition must be provided by the user
Integer sessionId();
#endif

template <class T>
T& Singleton<T>::instance() {
    static map<Integer, shared_ptr<T> > instances_;
    #if defined(QL_ENABLE_SESSIONS)
    Integer id = sessionId();
    #else
    Integer id = 0;
    #endif
    shared_ptr<T>& instance = instances_[id];
    if (!instance)
        instance = shared_ptr<T>(new T);
    return *instance;
}
```

このクラスは、Curiously Recurring Template Pattern をベースに構築しています。このパターンについては、既に Chapter VII で解説しています。あるクラスのインスタンスが Singleton（インスタンスの存在を1つしか許さない）である為には、そのクラス `C` を `Singleton < C>` クラステンプレートから派生させる必要があります。Singleton は private で宣言されたコンストラクターを提供し、その派生クラスは `Singleton < C>` をフレンドクラスとして宣言することによって、その private なコンストラクターを使用できます。この仕組みについては、`Settings` クラスの実装例をご覧下さい。

Scott Meyers（巻末 (Meyers, 2005) 参照）が提言している通り、Singleton のインスタンス（訳注：原文では instances と複数になっている）は、`instance()` メソッドの中で static 変数として宣

言されている map<Integer, shared_ptr<C> > オブジェクトに保持されます（Singleton なのに、なぜ複数のインスタンスを想定しているかの説明は後ほど）。こうする事によって、いわゆる static initialization order fiasco（静的オブジェクトの初期化順序の失敗。static オブジェクト（変数）が、それが初期化される前に他のオブジェクトで使用される時に発生する）の問題を回避し、さらに C++11 の環境では、そのオブジェクトの初期化は thread-safe になる事が保証されています。（後で見る通り、これで問題がすべて終わりという訳ではありませんが）

さて、読者の方はいくつかの疑問を持たれたかと思います。例えば、Singleton であるはずのインスタンスを、なぜインスタンスの map< > とするのか?（訳注：たった一つのインスタンスしか必要としないのに、なぜ複数のインスタンスを想定している map< > を使ってインデックスと対応付けているのか?）　実は、1つのインスタンスしか持たないと、それによる制約が発生するからです。例えば、ユーザーが、同時に異なる評価日を使って時価計算を行いたいと考えるかもしれません（訳注：時価評価日”EvaluationDate” クラスは、Singleton として宣言されている）。そういった場合への対応として、我々はスレッド毎にひとつの Singleton を持たせる事を許容しました。その選択は、コンパイル時のフラッグにより設定できるようになっており、instance() メソッドの中の #if 文による分岐を使って sessionID() 関数から、スレッド ID を取ってくるようにし、それを map<Integer, shared_ptr<C> > におけるインデックス用整数として使っています。コンパイル時フラッグで、スレッド毎の Singleton インスタンスの許容を選択した場合、ユーザー自身で sessionID() 関数を実装しなければなりません。その関数は、おそらく次のようなかたちになるでしょう。

```
Integer sessionId() {
    return /* some unique thread id from your system API */ ;
}
```

この関数の中で、ユーザーは OS が提供している関数（あるいはスレッドを管理用の Library が提供する関数）を使ってスレッドを特定し、そのスレッド ID を整数に変換して返すようにします。その結果、instance() メソッドは各スレッドに対応する Singleton インスタンスを返します。もしユーザーが、この機能をオフにすると、スレッド ID は常に 0 となり、instance() メソッドは常に同じ Singleton インスタンスを返します。その場合は、（問題含みなので）ユーザーはマルチスレッドを使いたいとは思わないでしょう。仮に使うとしても、よく注意する必要があります。理由は、前の Global Setting の説明をご覧下さい。

他にも、読者の方は、私が上記の Listing を“デフォールトの実装内容”と述べたのはなぜかと思われるかも知れません。いい所に気づかれました。ここでは詳しく説明しませんが、実は他にも Singleton クラスの実装方法が複数あり、これもコンパイル時のフラッグ設定で選択可能です。

ひとつは、（Singleton インスタンスを保持する）map< > オブジェクトを、instance() メソッドの中で宣言される static 変数とするのでは無く、Singleton クラスの static メンバー変数として宣言する方法です（QL_MANAGED フラッグを 1 に設定すればこうなります）。これは、map<> を instance() の中で宣言した場合 .NET フレームワークの中で managed C++ としてコンパイルされると、うまく機能しない事が判明したからです（少なくとも古いバージョンの Visual Studio のコンパイラーではそうです）。

もうひとつの方法は、ユーザーがマルチスレッド環境の中でグローバルな Singleton インスタンスを1つだけ使いたい場合に使います。その場合、Singleton インスタンスの初期化は thread safe

(マルチスレッド下でも安全にデータ共有できる状態)でなければなりません(ここでは Singleton インスタンスそのものの話をしており、それを保持している `map<Integer, T>` の事ではありません。それは `new T` で生成されます)。従って、この方法を選択した場合は、(thread safe を実現する為に) lock や mutex クラスなどが必要になります。デフォールトのシングルスレッドの設定の中に、これらの部品に関する記述を含めるべきではありません。従って、そのコードは、別のコンパイル時フラッグ設定の後ろにあります。興味のある方は、QuantLib ライブラリーのコードを見て下さい。

読者の方の最後の疑問点は、はたして Singleton クラスがそもそも必要だったのかどうかという事でしょう。それは厳しい質問です。私見については、前に説明した Global Settings の所を再度参照して下さい。今の所、Winston Churchill による民主主義制度の評価と同じ様に、悪い選択肢の中で最もましなものではないかと思っています。

11.7.3 The Visitor Pattern: ビジターパターン

QuantLib での Visitor Pattern の実装例を、次の Listing A-29 で示しますが、これは4人組の本による本来の形ではなく、Acyclic Visitor Pattern (巻末(Martin, 1997) 参照)を取り入れています。そこではまず、インターフェース用の degenerate な(訳注:すなわちメソッドを持たず何の動作もしない)`AcyclicVisitor` クラスを定義します。それと同時に(具体的な動作を担う)`Visitor` クラステンプレートを定義して、その中でテンプレート引数用の動作を提供する `visit()` を純粋仮想関数として宣言しています(訳注:従って `visit()` の具体的な動作内容は派生クラスで実装される)。

Listing A.30: `AcyclicVisitor` クラスと `Visitor` クラステンプレートの定義

```
class AcyclicVisitor {
  public:
    virtual ~AcyclicVisitor() {}
};

template <class T>
class Visitor {
  public:
    virtual ~Visitor() {}
    virtual void visit(T&) = 0;
};
```

このパターンを機能させるには、Visitor クラスの訪問を受ける側のクラス階層(訳注:Acceptor クラス、下記コードでは Event、CashFlow、Coupon クラスが該当)も必要になります。下記 Listing A.31 に、その一例を示します。

Listing A.31:`Visitor` から訪問を受ける側のクラスにおける `accept()` メソッドの実装内容

```
void Event::accept(AcyclicVisitor& v) {
    Visitor<Event>* v1 = dynamic_cast<Visitor<Event>*>(&v);
    if (v1 != 0)
        v1->visit(*this);
    else
        QL_FAIL("not an event visitor");
}

void CashFlow::accept(AcyclicVisitor& v) {
    Visitor<CashFlow>* v1 =
        dynamic_cast<Visitor<CashFlow>*>(&v);
    if (v1 != 0)
        v1->visit(*this);
    else
        Event::accept(v);
}

void Coupon::accept(AcyclicVisitor& v) {
    Visitor<Coupon>* v1 = dynamic_cast<Visitor<Coupon>*>(&v);
    if (v1 != 0)
        v1->visit(*this);
    else
        CashFlow::accept(v);
}
```

訪問を受ける側 (Acceptor) のクラス階層における各クラスは（少なくとも、Visitor クラスに何等かの動作を委託しているクラスは）、`AcyclicVisitor` の参照を引数として取る `accept()` メソッドを定義する必要があります。それぞれの `accept()` メソッドは、引数として渡された `AcyclicVisitor` インスタンスを、`dynamic_cast` を使って、自分自身のクラスに対応する（訳注:`Visitor<C>` のテンプレート引数 `C` に、自分自身の型を指定して具体化されている）Visitor インスタンスに型変換します。その型変換がうまくいけば、その Visitor インスタンスの `visit()` メソッドを呼び出します。型変換がうまくいかない場合は、代替策を探さなければなりません。仮にその Acceptor クラスが派生クラスであれば、そのベースクラスで実装されている `accept()` を呼び出し、そこで型変換を試みます。もしベースクラスでもうまく行かなかった場合は、例外処理に進みます（あるいは何もせずにメソッドを終了するという選択肢もありますが、失敗に気が付かないで終わるのは避けました）。（訳注：上記コード例では階層の一番下にある `Coupon` クラスの `accept()` 内で適合する Visitor を探し、それが存在しない場合親クラスである `CashFlow` クラスの accept( ) を呼び出して適合する Visitor を探し、さらにそれもうまくいかなければベースクラスである `Event` クラスの `accept()` を呼び出す。それでもダメな場合は例外処理に移る）

Visitor パターンは、Chapter IV のキャッシュフロー分析の所で説明した `BPSCalculator` クラスで取り入れられています。このクラスは `AcyclicVisitor` クラスから派生しており、従って様々なク

ラスの accept( ) メソッドの引数として使う事が出来ます。また同時に、テンプレート引数が特定され具体化された `Visitor<C>` からも派生させています（そのクラスでは `visit()` が実装されているはずです）。その Visitor インスタンスは、（それを必要とする）何等かのオブジェクトの `accept()` メソッドに渡され、そこで（型変換が）うまくいけば `visit()` メソッドが呼び出され、だめなら例外処理に飛びます。

Visitor パターンの有用性については、Chapter IV で既に説明したので、そこを参照して下さい。ここでは、なぜ AcyclicVisitor パターンを選択したかについて、2点ほど説明したいと思います。

端的に言えば、4人組が提唱した元々の Visitor パターンの方が、若干プログラムのスピードは速いかも知れませんが、より取扱いが厄介です。特に、訪問を受ける側（Acceptor 側）の階層に新たなクラスを追加した場合、それに対応する `visit()` メソッドを、既存の Visitor クラスの階層すべてについて追加する必要があります（そうしないとコードをコンパイルできません）。AcyclicVisitor パターンを使えば、その必要はありません。ユーザー定義の Acceptor 派生クラスに対応する Visitor クラスが定義されていなくても、`accept()` メソッドにおいて型変換が失敗して例外処理に飛ぶだけです。（注：実際の所、ユーザーは `accept()` そのものを定義する必要さえありません。ベースクラスのそれを継承すればいいのです。しかしその場合は、Visitor 側で、その Acceptor 用の `visit()` メソッドを実装しても意味がありません）

注意しておきますが、この事は（人生において多くの事がそうであるように）便利ではあるものの、良い事ではありません。新しい Acceptor をユーザー定義した場合でも、ユーザーは常に対応する Visitor クラスの階層を見直して、既存の `visit()` の実装を流用する事が問題ないかチェックし、もしそうでないなら新たに `visit()` を実装しなければなりません。しかし、コンパイラーがコンパイルできないデメリットと比べれば、`visit()` を全ての Acceptor クラス用の Visitor 階層に実装しないで済むメリットの方が大きいと思います。`BPSCalculator` クラスで言えば、様々な階層の CashFlow クラスに対応する BPS 計算アルゴリズムの実装が、2種類程度の `visit()` メソッドの実装で十分まかなえます。

12. Appendix B: プログラムコードの表記方法の慣行

どんなプログラマーの集団においても、いくつかのコードを書く conventions（慣行）を持っています。どのような conventions であっても（6〜7種類あり、数えきれない位の争いの原因になっています）、それに厳格に従う事が重要です。（注：但し、QuantLib ライブラリーの開発者も人間ですので、時々そのルールを守っていない事もあります）そうする事によって、プログラムコードの理解を容易にします。conventions に馴染みのある読者であれば、一目見ただけでマクロと関数の違いが判りますし、変数と変数型の名前の違いも判ります。

下記の Listing B.1 に QuantLib ライブラリーの中で使われている conventions を簡単に示します。Sutter and Alexandrescu, 2004 のアドバイスに従い、我々は、その数を最小限にし、読みやすさを向上させる conventions にのみ限定しようと努めました。

Listing B.1 ：QuantLib code conventions.の概要

```
#define SOME_MACRO

typedef double SomeType;

class SomeClass {
  public:
    typedef Real* iterator;
    typedef const Real* const_iterator;
};

class AnotherClass {
  public:
    void method();
    Real anotherMethod(Real x, Real y) const;
    Real member() const;   // getter, no "get"
    void setMember(Real);  // setter
  private:
    Real member_;
    Integer anotherMember_;
};

struct SomeStruct {
    Real foo;
    Integer bar;
```

```
};

Size someFunction(Real parameter,
                  Real anotherParameter) {
    Real localVariable = 0.0;
    if (condition) {
        localVariable += 3.14159;
    } else {
        localVariable -= 2.71828;
    }
    return 42;
}
```

マクロはすべて大文字で書かれ、単語の間を underscore(_)で分けています。データ型(クラスや構造体など)の名前は、大文字でスタートし、いわゆる camel 体(複合語の場合は各単語の先頭を大文字にする)で書かれています。上記例では `SomeType` のようなデータ型宣言や、`SomeClass` や `AnotherClass` といったクラス名がそうなっています。しかし、データ型宣言においては、C++ の標準ライブラリーに見られるような方法を真似て、違う方法をとる場合もあります。その例として、`SomeClass` の中の、2つの iterator 型の型宣言がそうなっています(訳注:先頭文字も含めて、すべて小文字になっている)。同じような例外が、内部クラスについても存在しています。

その他の殆どは(変数、関数、メソッド名、パラメータなど)小文字でスタートする camel 体で書かれています。クラスのメンバー変数も同じ慣行に従っていますが、最後に underscore (_) を付け足しています。そうする事によって、メソッドの中で使われるローカル変数との区別が容易になります(構造体やクラスの public なメンバー変数には例外がよく見られます)。メソッドの中で、getters メソッド(メンバー変数を取りだすメソッド)と setters(メンバー変数を書き込むメソッド)の間で、さらに特別な使い分け方法があります。setters メソッドでは、設定しようとするメンバー変数名の前に set を付け加え最後の underscore を取り除いた単語をメソッド名とし、getters メソッドでは取りだす変数名の最後の underscore を取り除いた単語をメソッド名としています(get は付け足しません)。この方法は、上記コードの `AnotherClass` や `SomeStruct` の中で例が確認できます。

かなり緩やかな慣行として、関数宣言や `if`、`else`、`for`、`while`、`do` の後の左かっこ(はそれらの keyword と同じ行に書くようにしています(上記コードの `someFunction()` がその例です)。さらに、`else` keyword は、その前に来る右かっこ)と同じ行に書くようにしています。同じ方法は、`while` 文の後に来る `do` 宣言でも取られています。しかし、この方式は、読みやすさの為と言うより、好みの問題と言えるので、開発者の方は、この方法が嫌であれば、自分流で書いても構わないでしょう。ここにのせている関数は、読みやすさを向上させる他の慣行例も載せています。それは、関数やメソッドの引数が、同じ行に書ききれない場合は、縦に並べて書くようにしています。

最後ですが大事なルールとして、コードの中で、tab を使わないようにしています。その代わり4文字分のスペースを使っています。もし、ここにある慣行の内、一つだけ従うとするなら、このルールにして下さい。そうしないと、大抵の場合、他のコードエディターを使っている開発者からすると、インデントが全くうまくいっていないコードに見えてしまうでしょう。読者の方がどのようなコードエディターを使っていても、4文字スペースの方法であればうまく調整されて、読者自身が手間

をかけなくともきちんと適用されているはずです。

13. QuantLib license

QuantLib is

- © 2000, 2001, 2002, 2003 RiskMap srl
- © 2001, 2002, 2003 Nicolas Di Césaré
- © 2001, 2002, 2003 Sadruddin Rejeb
- © 2002, 2003, 2004 Decillion Pty(Ltd)
- © 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2014, 2015 Ferdinando Ametrano
- © 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2014, 2016, 2017, 2018, 2019 StatPro Italia srl
- © 2003, 2004, 2007 Neil Firth
- © 2003, 2004 Roman Gitlin
- © 2003 Niels Elken Sønderby
- © 2003 Kawanishi Tomoya
- © 2004 FIMAT Group
- © 2004 M-Dimension Consulting Inc.
- © 2004 Mike Parker
- © 2004 Walter Penschke
- © 2004 Gianni Piolanti
- © 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019 Klaus Spanderen
- © 2004 Jeff Yu
- © 2005, 2006, 2008 Toyin Akin
- © 2005 Sercan Atalik
- © 2005, 2006 Theo Boafo
- © 2005, 2006, 2007, 2009 Piter Dias
- © 2005, 2013 Gary Kennedy
- © 2005, 2006, 2007 Joseph Wang
- © 2005 Charles Whitmore
- © 2006, 2007 Banca Profilo S.p.A.
- © 2006, 2007 Marco Bianchetti
- © 2006 Yiping Chen
- © 2006 Warren Chou
- © 2006, 2007 Cristina Duminuco
- © 2006, 2007 Giorgio Facchinetti
- © 2006, 2007 Chiara Fornarola

14. Bibliography

D.Abrahams, *Want Speed? Pass by Value*. In *C++ Next*, 2009.

D.Adams, *So Long, and Thanks for all the Fish*. 1984.

A.Alexandrescu, *Move Constructors*. In *C/C++ Users Journal*, February 2003.

F.Ametrano and M.Bianchetti, *Everything You Always Wanted to Know About Multiple Interest Rate Curve Bootstrapping but Were Afraid to Ask*. SSRN working papers series n.2219548, 2013.

J.Barton and L.R.Nackman, *Dimensional Analysis*. In *C++ Report*, January 1995.

T.Becker, *On the Tension Between Object-Oriented and Generic Programming in C++*. 2007.

Boost C++ libraries.http://boost.org.

M.K.Bowen and R.Smith, Derivative formulae and errors for non-uniformly spaced points. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 461, pages 1975–1997. The Royal Society, 2005.

D.Brigo and F.Mercurio, *Interest Rate Models —Theory and Practice*, 2nd edition. Springer, 2006.

Mel Brooks (director), *Young Frankenstein*. Twentieth Century Fox, 1974.

W.E.Brown, *Toward Opaque Typedefs for C++1Y, v2*. C++ Standards Committee Paper N3741, 2013.

L.Carroll, *The Hunting of the Snark*. 1876.

G.K.Chesterton, *Alarms and Discursions*. 1910.

M.P.Cline, G.Lomow and M.Girou, *C++ FAQs*, 2nd edition. Addison-Wesley, 1998.

J.O.Coplien, *A Curiously Recurring Template Pattern*. In S.B.Lippman, editor, *C++ Gems*. Cambridge University Press, 1996.

C.S.L.de Graaf. *Finite Difference Methods in Derivatives Pricing under Stochastic Volatility Models*. Master's thesis, Mathematisch Instituut, Universiteit Leiden, 2012.

C.Dickens, *Great Expectations*. 1860.

P.Dimov, H.E.Hinnant and D.Abrahams, *The Forwarding Problem: Arguments*. C++ Standards Committee Paper N1385, 2002.

M.Dindal (director), *The Emperor's New Groove*. Walt Disney Pictures, 2000.

D.J.Duffy, *Finite Difference Methods in Financial Engineering: A Partial Differential Equation Approach*. John Wiley and Sons, 2006.

M.Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd edition.

Addison-Wesley, 2003.

M.Fowler, *Fluent Interface*. 2005.

M.Fowler, K.Beck, J.Brant, W.Opdyke and D.Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

E.Gamma, R.Helm, R.Johnson and J.Vlissides, *Design Patterns: Element of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

P.Glasserman, *Monte Carlo Methods in Financial Engineering*. Springer, 2003.

D.Gregor, *A Brief Introduction to Variadic Templates*. C++ Standards Committee Paper N2087, 2006.

H.E.Hinnant, B.Stroustrup and B.Kozicki, *A Brief Introduction to Rvalue References*. C++ Standards Committee Paper N2027, 2006.

A.Hunt and D.Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.

International Standards Organization, *Programming Languages —C++*. International Standard ISO/IEC 14882:2014.

International Swaps and Derivatives Associations, *Financial products Markup Language*.

P.Jäckel, *Monte Carlo Methods in Finance*. John Wiley and Sons, 2002.

J.Kerievsky, *Refactoring to Patterns*. Addison-Wesley, 2004.

R.Kleiser (director), *Grease*. Paramount Pictures, 1978.

H.P.Lovecraft, *The Call of Cthulhu*. 1928.

R.C.Martin, *Acyclic Visitor*. In *Pattern Languages of Program Design 3*. Addison-Wesley, 1997.

S.Meyers, *Effective C++*, 3rd edition. Addison-Wesley, 2005.

N.C.Myers, *Traits: a new and useful template technique*. In The C++ Report, June 1995.

G.Orwell, *Animal Farm*. 1945.

W.H.Press, S.A.Teukolsky, W.T.Vetterling and B.P.Flannery, *Numerical Recipes in C*, 2nd edition. Cambridge University Press, 1992.

QuantLib. http://quantlib.org.

E.Queen, *The Roman Hat Mystery*. 1929.

V.Simonis and R.Weiss, *Exploring Template Template Parameters*. In *Perspectives of System Informatics*, number 2244 in Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2001.

B.Stroustrup, *The C++ Programming Language*, 4th edition. Addison-Wesley, 2013.

H.Sutter, *You don't know const and mutable*. In Sutter's Mill, 2013.

H.Sutter and A.Alexandrescu, *C++ Coding Standards*. Addison-Wesley, 2004.

T.Veldhuizen, *Techniques for Scientific C++*. Indiana University Computer Science Technical Report

TR542, 2000.

H.G.Wells, *The Shape of Things to Come*. 1933.

P.G.Wodehouse, *My Man Jeeves*. 1919.