

Getting Started with QuantLib

by [Vasily Nekrasov](#)

Preface

Currently there are many books on quantitative finance for graduate students. Most of them are pretty similar and repeat the same stuff on risk-neutral valuation (Black-Scholes model, no-arbitrage, Ito calculus, martingale measure, etc). There are also enough books on numerical finance. These books are more individual, some concentrate on theoretical aspects (convergence, computational efficiency), some focus on programming aspects and on C++ in particular.

This book is different. First of all it is written with practice in mind: **bitter experience clearly tells us that a *working* quant not necessarily needs to understand all mathematical nuances¹ but must be able to provide a proper final result and do it *fast*.** The main goal of this book is to teach you this. Another important goal is to give you some hand-on experience with *big* software projects like QuantLib. To my best knowledge, there is no such other book. Additionally, you will also get familiar with version control, doxygen, unit tests and so on. Formally this stuff is not a programming (and thus rarely taught in lectures on programming) but every software engineer needs it in practice².

I deliberately decided to stick to Windows and Visual Studio and not to discuss programming under Linux and Eclipse. These are important issues but the quants usually develop under Windows³ and porting to Linux (if necessary) is usually assigned to "pure IT" staff. Moreover, Microsoft has recently liberalized the distribution policy and everyone can get Visual Studio Community Edition free of charge⁴. Additionally, students may download VS Professional for free (note that Community and Professional editions seem to be essentially the same). Interestingly, QuantLib was initially Windows-only project and porting to Linux was one of the first community contribution⁵.

Since I am most experienced as fixed income quant, we will mostly concentrated on respective instruments and interest rate models. This is not due to my personal caprice but rather due to the fact that virtually all financial (and many non-financial) institutions have interest rate swap[tion]s in their portfolios. Moreover, starting with even simplest stock models (that imply stochastic processes) may be hard and starting with the calculation of a bond yield shall be easy (however, not as easy as you might believe). Interestingly, the fathers of QuantLib got started at the interest rate desk⁶.

Writing this book I tried to keep the prerequisites as moderate as possible. I assume the basic knowledge of C++ and object oriented programming (the latter is very briefly reviewed). If you do not possess this stuff, you should first read e.g. "C++ from the Ground Up" by Herbert Schildt. On the other and I do not assume any knowledge of design patterns. Those that are relevant for QuantLib, will be discussed in detail.

As to stochastic finance, as long as we deal with date and bond arithmetic, you do not need it. In the sense of 80–20 Pareto rule⁷ in may be clever for a "pure programmer" to learn only this relatively easy stuff in order to increase the chances to get a job in a bank or corporate treasury. For more advanced topics some

¹ Though it never hurts and often is still useful.

² Probably, the knowledge of this stuff (or lack of it) distinguishes a programmer from a code monkey.

³ Likely because their solutions are often integrated in Excel.

⁴ <http://www.visualstudio.com/en-us/news/vs2013-community-vs.aspx>

⁵ <http://www.moneyscience.com/pg/blog/Admin/read/464541/open-source-finance-1-quantlib-an-interview-with-luigi-ballabio>

⁶ *ibid*

⁷ http://en.wikipedia.org/wiki/Pareto_principle

knowledge of stochastic finance is required. Ideally, you should first read both volumes of "Stochastic Calculus for Finance" by Steven Shreve and then "Arbitrage Theory in Continuous Time" by Tomas Björk. If you want to quickly get your hands dirty, read "Financial Calculus: An Introduction to Derivative Pricing" by Baxter and Rennie. This book omits many details but provides a sufficient overview for lazybones. And those, who want to dig deeply and dwell on details should have a look at my LIBOR market model tutorial in Appendix B.

Finally, it is worth telling how I got started with programming and with QuantLib. Well, my first experience was with Basic on KOPBET (Russian 8-bit computer) in the age of 12. Then I learnt Turbo Pascal on i286 in school and Z80 assembler on ZX Spectrum. In 1997 my family could to the first time afford a "serious" computer (it was P166 MMX). By that time the Internet came to Russia and I quickly learnt web programming by myself (at first Perl then PHP) and (being a student) I moonlighted as a web developer. In 1999 I got started with Java ... which turned out to be pretty hard. Object oriented programming was something radically new. Additionally, I encountered event driven model, multithreading and so on. The literature was scarce by that time and the most of freely available tutorials were obsolete and considered Java 1.0, whereas I got started with Java 1.1 and then Java 1.2...

As a result, I was able to write simple applets but it took a couple of years before I *really* understood the OOP. I also read a couple of books on C++ but did not really used it. After I graduated from the university in 2002 I worked as web developer till I moved to Germany in 2005 to make my Master of financial engineering. University programs in Germany are usually very theoretic and mine was no exception. So I dwelt upon theory (including fine measure-theoretic aspects), which I do not regret since if you miss it in the university, you will hardly get a chance to catch up. But since I got few computational tasks, the programming was neglected. Only as I started with my master thesis I revived my programming skills because both theoretical and numerical results were required. By that time I heard about QuantLib (version 0.8.0). I attempted several time to build it from sources and finally got it. However, the installation of the boost binaries was only partially successful and the unittest suite did not work. Analogously to my Java case, I encountered too many new stuff with QuantLib: boost libraries, design patterns, date arithmetic... but now I was very short of time and thus postponed it for a while.

After completing my thesis I got a job as quant developer. I recalled and complemented my C++ knowledge. My main duty was the maintenance and enhancement of a calculation engine ("Rechenkern" in German). This engine was primarily developed to calculate loans and mortgages but it also had some functions to price derivatives. So I used QuantLib to check the calculation results from this Rechenkern. I must note that by that time I did not deeply understood QuantLib. I merely looked how it is done in unit tests then often just put my values and got the result. The situation changed as Dimitri Reisch's excellent tutorials were published online. They gave me a holistic view of QuantLib.

Later I changed to the German Finance Agency and used QuantLib for model risk validation. I also wanted to implement the HPSn model in QuantLib⁸, which was developed by my front office colleagues. However, there was little interest to my initiative, so I just implemented a very alpha version that was really not worth checking-in to QuantLib repository.

Now, as you know my QuantLib experience, you will likely understand why I am not going to teach you how to contribute to QuantLib. This is a purpose of Luigi Ballabio's book⁹ and he definitely can achieve it better than me. I am going to teach you to use the already available QuantLib functionality (which is really huge) and how to write [quick and dirty] extensions that solve *your* problems.

⁸ <http://www.yetanotherquant.com/QuantLib/ImplementingHPS/>

⁹ <http://implementingquantlib.blogspot.de/p/the-book.html>

Chapter 1: Building QuantLib and setting up developer toolbox

Building QuantLib is not an easy task. I know enough quants that gave up QuantLib because they failed to [quickly] compile it from sources. Thus let us discuss the installation process in detail. First of all you need to install Visual Studio. By the time I wrote this book the current version was VS2013. I highly recommend you to install a community edition, which, as I have already said, is free. VS Express will also do but it does not support plug-ins like Visual Assist (see below).

As the next step you need to build boost libraries. These libraries are "one of the most highly regarded and expertly designed C++ library projects in the world."¹⁰ To build them do the following:

1. Download the source code from www.boost.org. Current version is 1.57.0. Unpack to C:\
2. Start Visual Studio, go to TOOLS -> Visual Studio Command Prompt (Figure 0.1)
3. Change directory to C:\boost_1_57_0\tools\build
4. Run bootstrap.bat (this will build b2.exe)
5. Copy b2.exe to C:\boost_1_57_0 and run it there (Figure 0.2)
6. Drink coffee (if you use older version of Visual Studio or boost, you might also have a lunch)

Note that boost evolves quickly and the build-tools also do. A couple of years ago bjam.exe was to be used but currently it is replaced with b2.exe. That's why always have a look at

<http://www.nuclearphynance.com/Show%20Post.aspx?PostIDKey=152032>

In this thread I will provide the current setup instructions for boost.

Now we may build QuantLib. Current version is 1.4.1, which supports VS2012 but not VS2013. Fortunately, it is not a problem, just open QuantLib_vc11.sln in VS2013 and it will automatically convert it¹¹ (Figure 0.3). Most likely, VS2013 will be officially supported when this book gets published.

As the next step you need to set up the boost path. In Solution Explorer select all projects, select Properties -> Configuration Properties -> VC++ Directories and add C:\boost_1_57_0 to Include Directories and C:\boost_1_57_0\stage\lib to Library Directories (Figure 0.4). Press F7 to build solution (you may again go drink coffee).

In principle, we can already start with QuantLib but from practical point of view we are not yet done with toolbox setup. First of all I highly recommend you to install the Visual Assist¹², a Visual Studio plug-in, which greatly enhances its functionality. It allows you to quickly switch between header (.hpp) and .cpp files (how may Microsoft have missed this feature?!), flexibly search for files in your project by keywords (Figure 1.5), find all references to a variable and much more. Visual Assist is not cheap but there are discounts for private users and students. But if you are a really poor student you may alternatively install VSAid¹³. At least it allows a quick switch between .hpp and .cpp, which is really indispensable. Note that neither Visual Assist nor VSAid work with Visual Studio Express.

Further you should install a **version control** system. I am a big fan of Subversion (aka SVN). QuantLib itself used SVN before migrating to Git. However, Git may mostly be advantageous for big "geographically dissipated" teams. For a "one-or-two developers team" SVN should be fine.

¹⁰ Herb Sutter and Andrei Alexandrescu, C++ Coding Standards

¹¹ Such straightforward conversion was not always the case. For example, as one wanted to convert QuantLib 1.2.1 from VS2010 to VS2012 a quick and dirty trick was to replace error "unknown Microsoft compiler" with define QL_LIB_TOOLSET "vc100" in D:\sandbox\QuantLib-1.2.1\ql\auto_link.hpp.

¹² <http://www.wholetomato.com/features/default.asp>

¹³ <http://www.brocksoft.co.uk/vsaid.php>

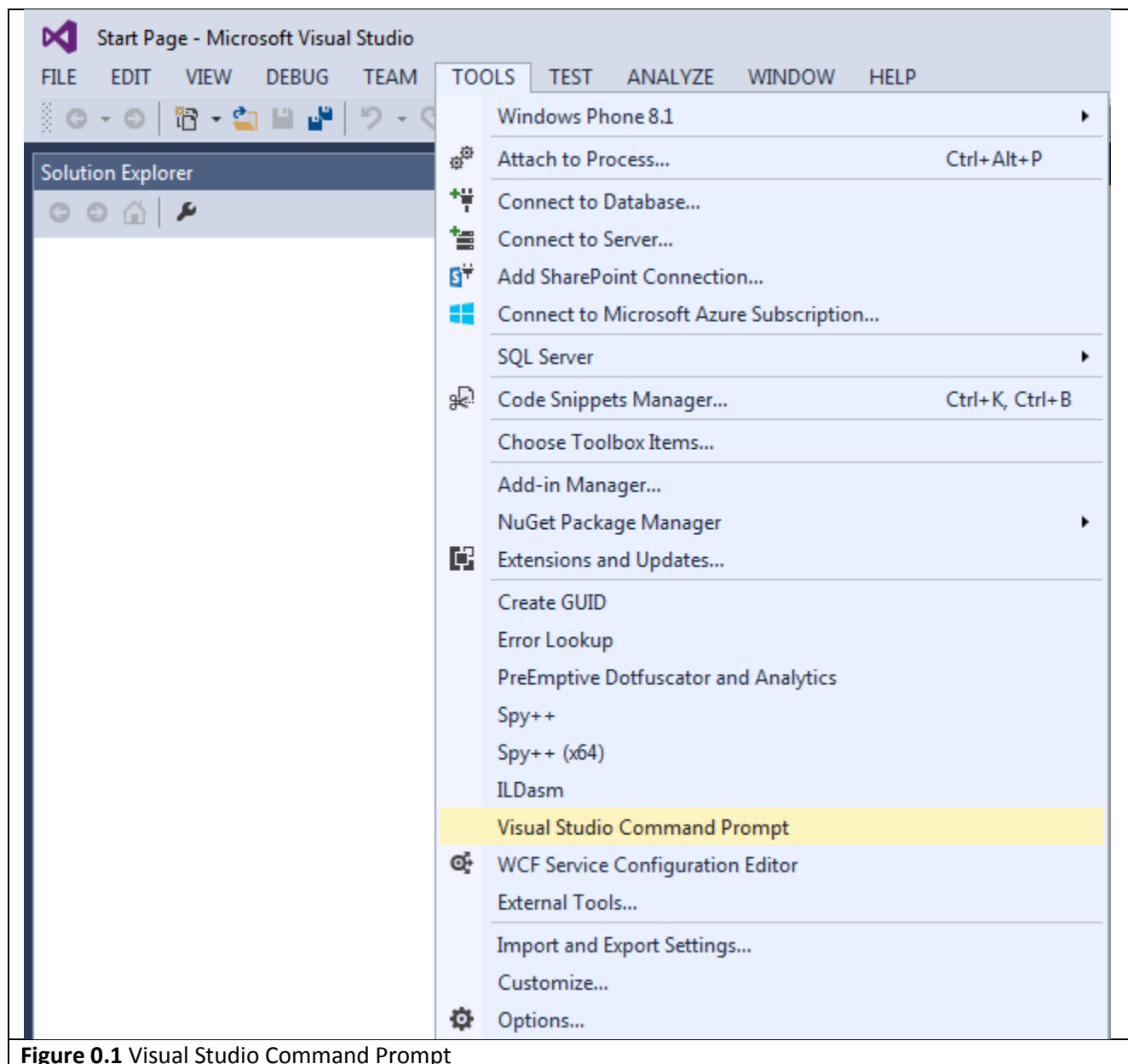


Figure 0.1 Visual Studio Command Prompt

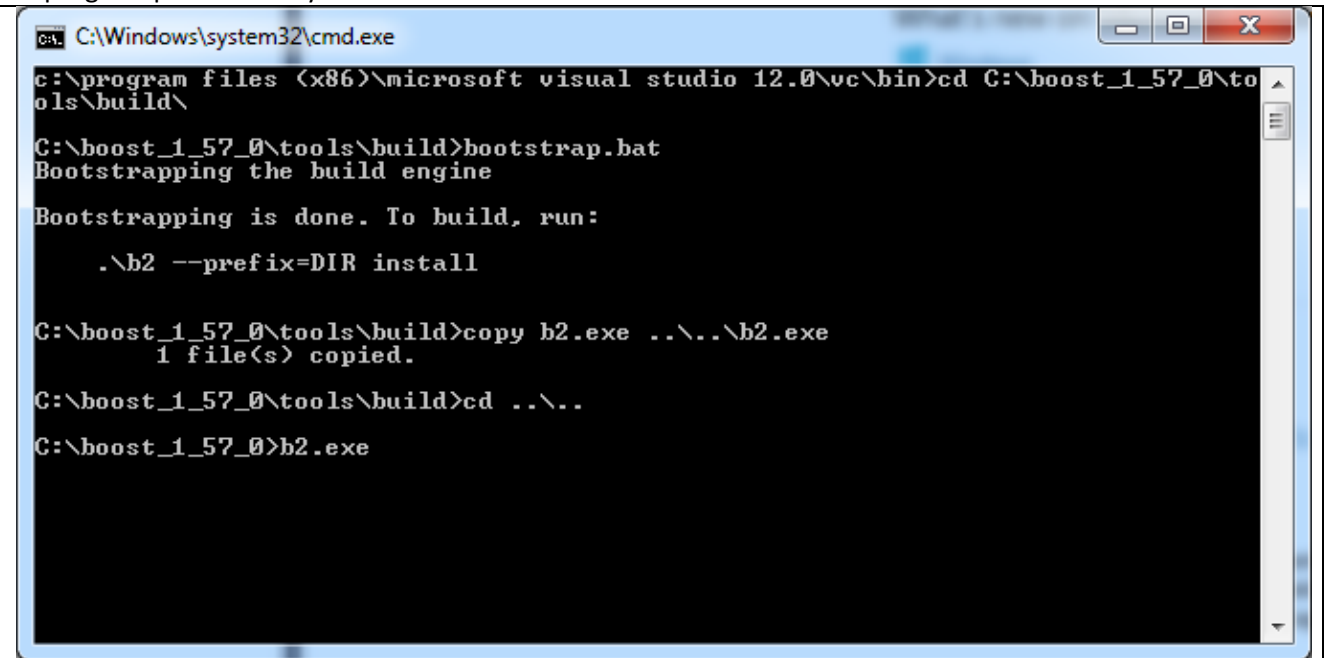
Unfortunately, I cannot teach you version control in detail. But there are enough tutorials in Internet. I just briefly tell you that a proper version control allows you to track the changes, compare versions, rollback to previous version if the current version is erroneous and much more. No serious software project can do without version control¹⁴. Writing this book I also use SVN: initially a version control was applicable to text files only, but now it can be used for MS Word documents as well (Figure 0.6) and even for the graphic data!

The easiest way to install SVN is to download respective stack from bitnami¹⁵. This is a server side part, which by default will be available under <http://localhost/subversion/>

¹⁴ Well, I know a bank, whose quants were advanced enough to use QuantLib but they had no version control due to highly bureaucratic IT policy by this bank.

¹⁵ <https://bitnami.com/stack/subversion>

Additionally, you need an SVN client, I highly recommend TortoiseSVN¹⁶. If you are confused, well, you may also work without a version control. But I highly recommend you to learn it, you *will* need it if you are going to program professionally!



```
C:\Windows\system32\cmd.exe
c:\program files (x86)\microsoft visual studio 12.0\vc\bin>cd C:\boost_1_57_0\tools\build\
C:\boost_1_57_0\tools\build>bootstrap.bat
Bootstrapping the build engine
Bootstrapping is done. To build, run:
    .\b2 --prefix=DIR install
C:\boost_1_57_0\tools\build>copy b2.exe ..\..\b2.exe
1 file(s) copied.
C:\boost_1_57_0\tools\build>cd ..\..
C:\boost_1_57_0>b2.exe
```

Figure 0.2 Building boost libraries from command line

¹⁶ <http://tortoisesvn.net>

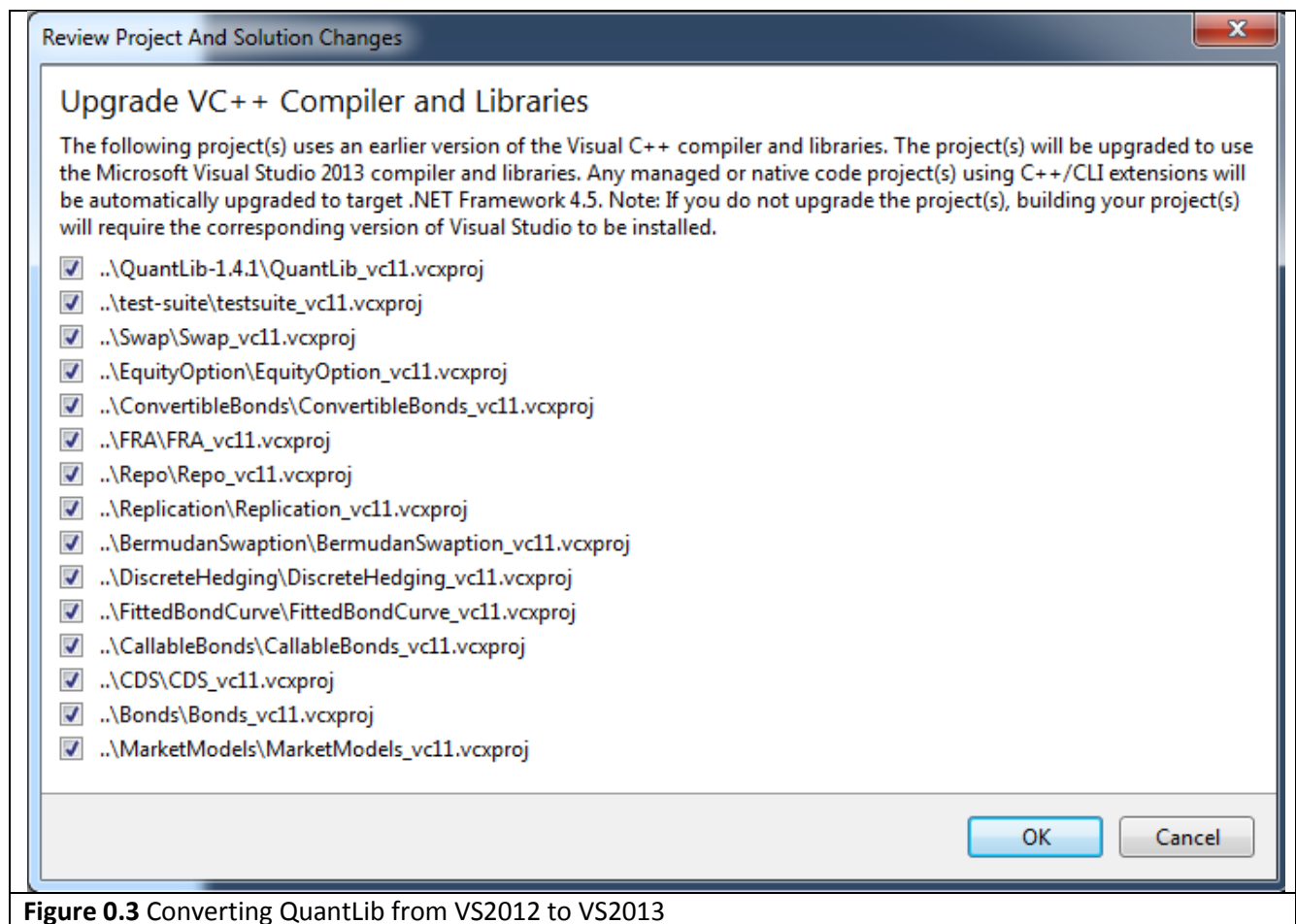


Figure 0.3 Converting QuantLib from VS2012 to VS2013

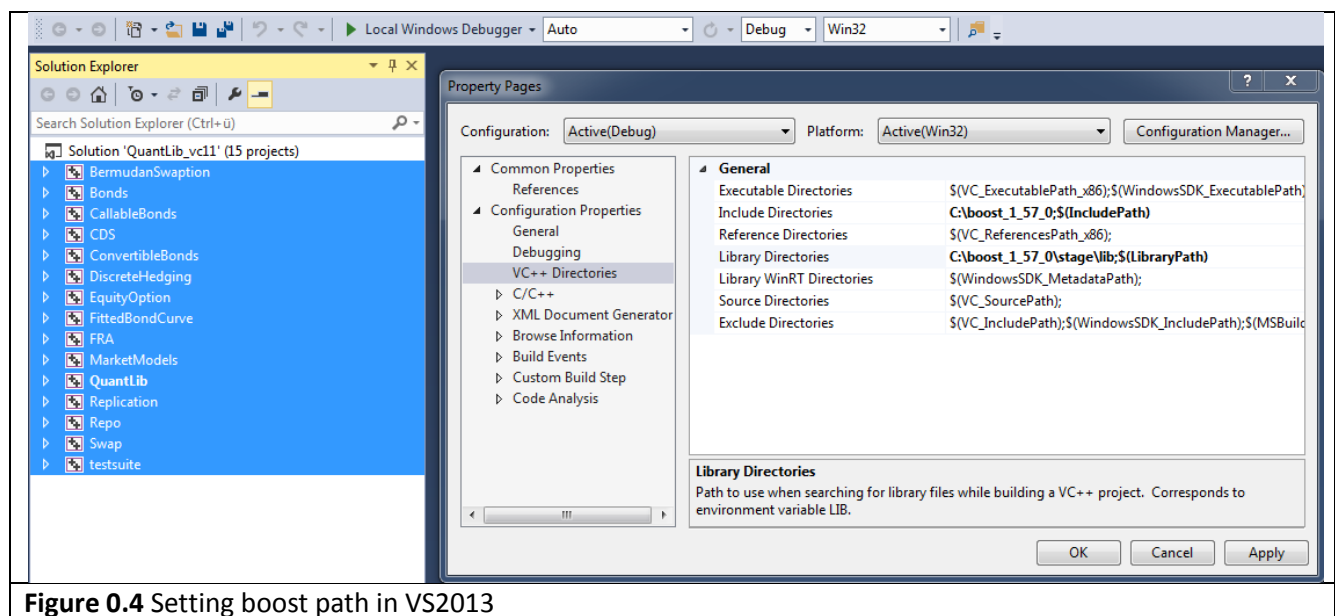


Figure 0.4 Setting boost path in VS2013

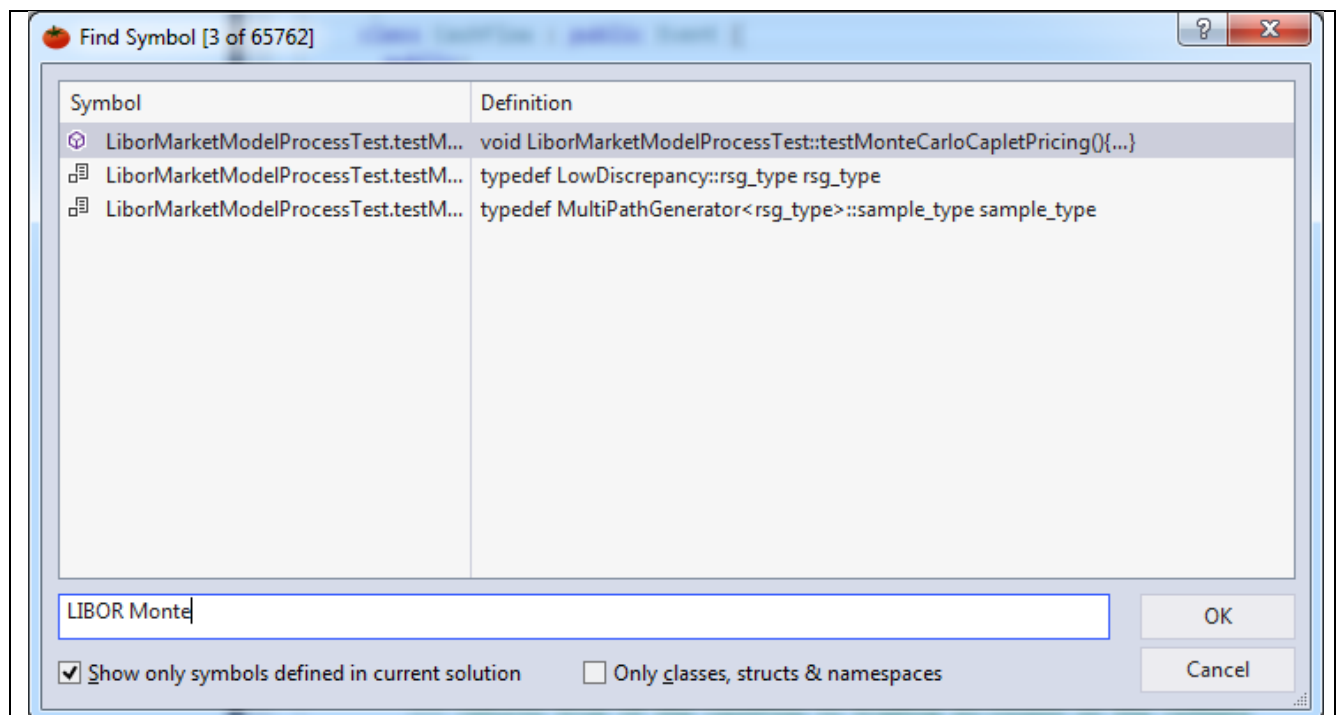


Figure 0.5 Visual Assist powerful search

Zusammenfassung: 6 Überarbeitungen

Einfügungen: 3 Löschvorgänge: 2 Verschiebungen: 0

Formatierung: 1 Kommentare: 0

Änderungen und Kommentare des Hauptdokuments

Eingefügt

Comparison

I am a big fan of Subversion (aka SVN). QuantLib itself used SVN before migrating to Git. However, Git may mostly be advantageous for big "geographically dissipated" teams. For a "one-or-two developers team" SVN should be fine.

Gelöscht

Comparison

Formatiert

Comparison

Block

Eingefügt

Comparison

Unfortunately, I cannot teach you version control in details. But there are enough tutorials in Internet. I just briefly tell you that a proper version control allows you to track the changes, rollback to previous version if the current version is erroneous and much more. No serious software project can do without version control¹⁴. Writing this book I also use SVN.

Gelöscht

Comparison

Änderungen der Kopf- und Fußzeile

(ohne)

Textfeldänderungen

(ohne)

Änderungen an Textfeldern in der Kopf- und Fußzeile

(ohne)

Fußnotenänderungen

Eingefügt

Comparison

¹⁴ Well, I know a bank, whose quants were advanced enough to use QuantLib but they had no version control due to highly bureaucratic IT policy by this bank.

Endnotenänderungen

(ohne)

Vergleichenes Dokument

4. Run bootstrap.bat (this will build b2.exe)

5. Copy b2.exe to C:\boost_1_57_0 and run it there (Figure 1.2)

6. Drink coffee (if you use older version of Visual Studio or boost, you might also have a lunch)

Note that boost evolves quickly and the build-tools also do. A couple of years ago bjam.exe was to be used but currently it is replaced with b2.exe. That's why always have a look at <http://www.nuclearphynance.com/Show%20Post.aspx?PostIDKey=152032>

In this thread I will provide the current setup instruction for boost.

Now we may build QuantLib. Current version is 1.4.1, which supports VS2012 but not VS2013. Fortunately, it is not a problem, just open QuantLib_vc11.sln in VS2013 and it will automatically convert it¹¹ (Figure 1.3). Most likely, VS2013 will be officially supported when this book gets published.

As the next step you need to set up the boost path. In Solution Explorer select all projects, select Properties -> Configuration Properties -> VC++ Directories and add C:\boost_1_57_0 to Include Directories and C:\boost_1_57_0\stage\lib to Library Directories (Figure 1.4). Press F7 to build solution (you may again go drink coffee).

In principle, we can already start with QuantLib but from practical point of view we are not yet done with toolbox setup. First of all I highly recommend you to install the Visual Assist¹², a Visual Studio plug-in, which greatly enhances its functionality. It allows you to quickly switch between header (.hpp) and .cpp files (how may Microsoft have missed this feature?!), flexibly search for files in your project by keywords (Figure 1.5), find all references to a variable and much more. Visual Assist is not cheap but there are discounts for private users and students. But if you are a really poor student you may alternatively install VSAid¹³. At least it allows a quick switch between .hpp and .cpp, which is really indispensable. Note that neither Visual Assist nor VSAid work with Visual Studio Express.

Further you should install a version control system. [I am a big fan of Subversion \(aka SVN\). QuantLib itself used SVN before migrating to Git. However, Git may mostly be advantageous for big "geographically dissipated" teams. For a "one-or-two developers team" SVN should be fine.](#)

Start Page - Microsoft Visual Studio

FILE EDIT VIEW DEBUG TEAM TOOLS TEST ANALYZE WINDOW

Solution Explorer

Windows Phone 8.1

Attach to Process...

Connect to Database...

Connect to Server...

Add SharePoint Connection...

Connect to Microsoft Azure Subscription...

SQL Server

Code Snippets Manager...

Choose Toolbox Items...

Add-in Manager...

NuGet Package Manager

Extensions and Updates...

Create GUID

Error Lookup

PreEmptive Dotfuscator and Analytics

Spy++

Spy++ (x64)

ILDasm

Visual Studio Command Prompt

WCF Service Configuration Editor

External Tools...

Import and Export Settings...

Customize...

Options...

Figure 1.1 Visual Studio Command Prompt

Unfortunately, I cannot teach you version control in details. But there are enough tutorials in Internet. I just briefly tell you that a proper version control allows you to track the changes, rollback to previous version if the current version is erroneous and much more. No serious software project can do without version control¹⁴. Writing this book I also use SVN.

C:\Windows\system32\cmd.exe

c:\program files (x86)\microsoft visual studio 12.0\vc\bin>cd C:\ols\build\

C:\boost_1_57_0\tools\build>bootstrap.bat

Bootstrapping the build engine

Bootstrapping is done. To build, run:

.\b2 --prefix=DIR install

C:\boost_1_57_0\tools\build>copy b2.exe ..\..\b2.exe

Originaldokument (BookQuantLib.docx-rev1.svn000.tmp)

4. Run bootstrap.bat (this will build b2.exe)

5. Copy b2.exe to C:\boost_1_57_0 and run it there (Figure 1.2)

6. Drink coffee (if you use older version of Visual Studio or boost, you might also have a lunch)

Note that boost evolves quickly and the build-tools also do. A couple of years ago bjam.exe was to be used but currently it is replaced with b2.exe. That's why always have a look at <http://www.nuclearphynance.com/Show%20Post.aspx?PostIDKey=152032>

In this thread I will provide the current setup instruction for boost.

Now we may build QuantLib. Current version is 1.4.1, which supports VS2012 but not VS2013. Fortunately, it is not a problem, just open QuantLib_vc11.sln in VS2013 and it will automatically convert it¹¹ (Figure 1.3). Most likely, VS2013 will be officially supported when this book gets published.

As the next step you need to set up the boost path. In Solution Explorer select all projects, select Properties -> Configuration Properties -> VC++ Directories and add C:\boost_1_57_0 to Include Directories and C:\boost_1_57_0\stage\lib to Library Directories (Figure 1.4). Press F7 to build solution (you may again go drink coffee).

In principle, we can already start with QuantLib but from practical point of view we are not yet done with toolbox setup. First of all I highly recommend you to install the Visual Assist¹², a Visual Studio plug-in, which greatly enhances its functionality. It allows you to quickly switch between header (.hpp) and .cpp files (how may Microsoft have missed this feature?!), flexibly search for files in your project by keywords (Figure 1.5), find all references to a variable and much more. Visual Assist is not cheap but there are discounts for private users and students. But if you are a really poor student you may alternatively install VSAid¹³. At least it allows a quick switch between .hpp and .cpp, which is really indispensable. Note that neither Visual Assist nor VSAid work with Visual Studio Express.

Überarbeitetes Dokument (BookQuantLib - Comparison)

4. Run bootstrap.bat (this will build b2.exe)

5. Copy b2.exe to C:\boost_1_57_0 and run it there (Figure 1.2)

6. Drink coffee (if you use older version of Visual Studio or boost, you might also have a lunch)

Note that boost evolves quickly and the build-tools also do. A couple of years ago bjam.exe was to be used but currently it is replaced with b2.exe. That's why always have a look at <http://www.nuclearphynance.com/Show%20Post.aspx?PostIDKey=152032>

In this thread I will provide the current setup instruction for boost.

Now we may build QuantLib. Current version is 1.4.1, which supports VS2012 but not VS2013. Fortunately, it is not a problem, just open QuantLib_vc11.sln in VS2013 and it will automatically convert it¹¹ (Figure 1.3). Most likely, VS2013 will be officially supported when this book gets published.

As the next step you need to set up the boost path. In Solution Explorer select all projects, select Properties -> Configuration Properties -> VC++ Directories and add C:\boost_1_57_0 to Include Directories and C:\boost_1_57_0\stage\lib to Library Directories (Figure 1.4). Press F7 to build solution (you may again go drink coffee).

In principle, we can already start with QuantLib but from practical point of view we are not yet done with toolbox setup. First of all I highly recommend you to install the Visual Assist¹², a Visual Studio plug-in, which greatly enhances its functionality. It allows you to quickly switch between header (.hpp) and .cpp files (how may Microsoft have missed this feature?!), flexibly search for files in your project by keywords (Figure 1.5), find all references to a variable and much more. Visual Assist is not cheap but there are discounts for private users and students. But if you are a really poor student you

Figure 0.6 SVN and MS Word: comparing two versions of this book

Last but not least we install the Doxygen¹⁷. "Doxygen is de facto the standard tool for generating documentation from annotated C++ sources ... [it] is very useful to *quickly find your way in large source distributions*. Doxygen can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams; all of them are generated automatically".

As I started my career as a quant developer, I had to maintain a big library like QuantLib. Though I previously had some experience with big projects, all of them were websites with a straightforward business logic, directly visible in a browser. Now it was not the case anymore. Fortunately, the library had a rich suite of unittests (QuantLib also has). So as I applied Doxygen to the source code, it extracted a lot of information. The most important were likely the inheritance and call/caller graphs. Let us, for example, have a look at the implementation of the Black-76 model in QuantLib. Doxygen has generated the following inheritance, call and caller graphs:

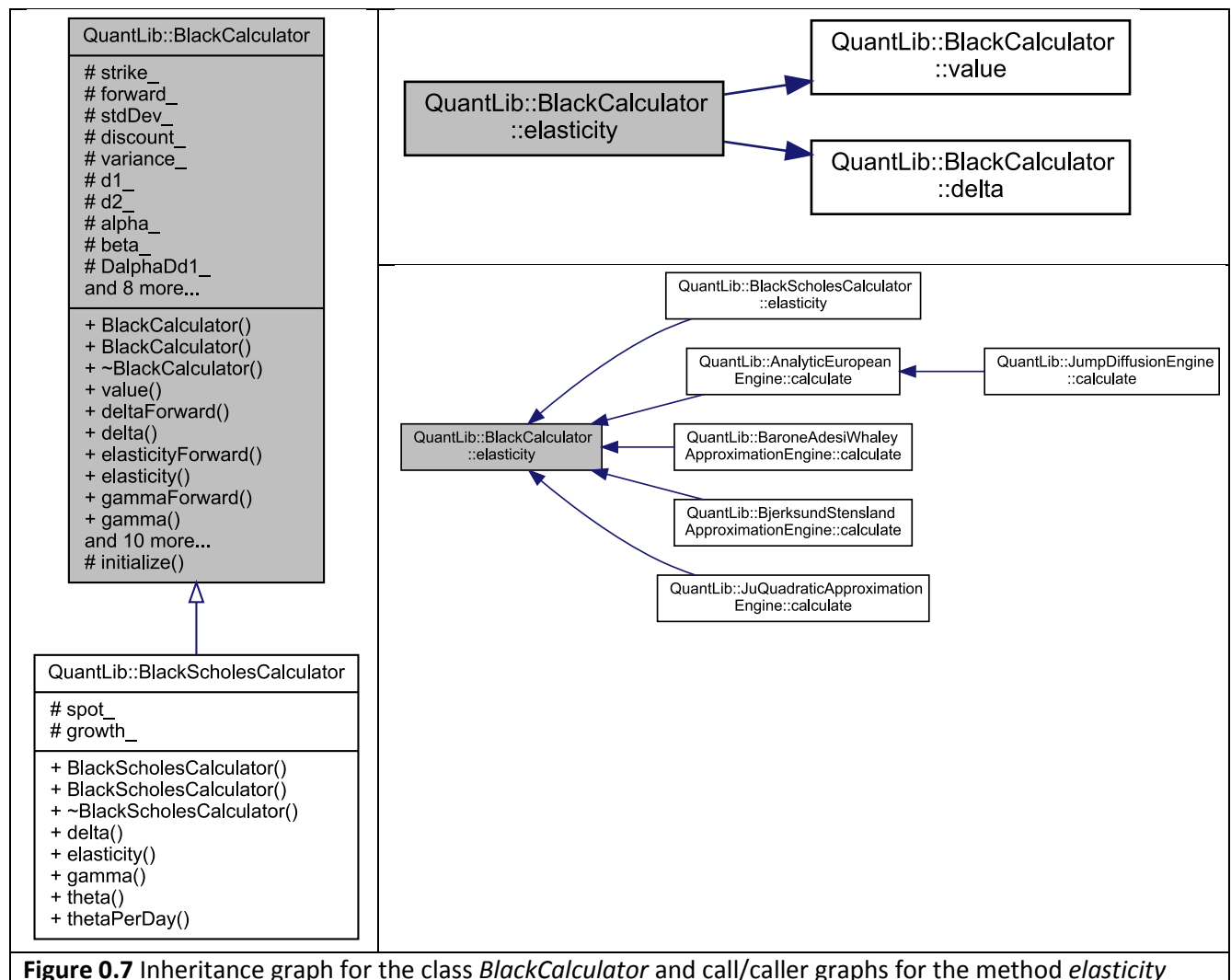


Figure 0.7 Inheritance graph for the class *BlackCalculator* and call/caller graphs for the method *elasticity*

From the inheritance graph we see that the class *BlackScholesCalculator* inherits from the class *BlackCalculator*. In particular, it means that if we modify the *BlackCalculator*, this will likely affect the *BlackScholesCalculator*. Call/caller graphs for the method *elasticity* tells us where this method is called and

¹⁷ <http://www.stack.nl/~dimitri/doxygen/>

which methods it calls. It may be very useful for debugging and we are aware, which methods can be affected if we change something in *elasticity*.

Doxygen has a built-in graph generator but if you want to have them generated nicely, you need to install the Graphviz¹⁸. It is also recommended to install LaTeX¹⁹. Doxygen understands LaTeX commands and they are to be found in QuantLib sources.

You should also tune the Doxygen settings in such a way that it generates full graphs. QuantLib developers, themselves, use Doxygen to generate official QuantLib HTML documentation. For the visibility's sake they limit the number of nodes in graphs. However, such visibility may be deceptive, for example, the inheritance graph for the Vasicek model from the official documentation neither tells us that Vasicek belongs to the realm of calibrated models²⁰ nor shows the members and functions of the classes. An obvious shortcoming of the complete graphs is that they can get illegible. But a 4K monitor²¹ can at least partially mitigate this problem.

Ok, we are (almost) done with technical setup, let us run some QuantLib code. Assign project *Bonds* as StartUp project, set a breakpoint at line 240 in Bonds.cpp and start debugging (F5).

Open *Bonds.cpp*, press Ctrl + G to and enter 240 in order to jump to the line 240

VS Tip

The variables representation in debugger window is not very clear. In particular, *today'sDate* is represented as *{serialNumber_=39706}* and the representation of all shared pointers *boost::shared_ptr<T>* (soon we will discuss them in detail) is hardly legible. We lay aside the former for a while but can quickly fix the latter. Just install the Natvis visualizer for boost by Arkady Shapkin from <https://visualstudiogallery.msdn.microsoft.com/61f12e7a-bc62-4b2c-b02e-d66014688c2e> run the same code again and compare the representation in debugger (Figure 0.9).

Natvis lets you customize the representation of C++ types in the debugger. It is a relatively new technology, which was introduced in VS2012. Before one overwrote the data representations in *autoexp.dat* (not supported anymore from VS2012). As a *quant* developer you probably should not dwell upon this technical details too much. But it is obviously worth being aware of this technology and installing ready-to-use visualizers, if available.

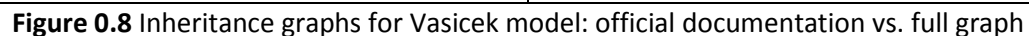
VS Tip

¹⁸ <http://www.graphviz.org>

¹⁹ <http://miktex.org>

²⁰ which means that QuantLib lets us fit the model parameters to the current term structure and a set of traded instruments like swaptions or caps.

²¹ 4K monitors have 3840 x 2160 pixels.



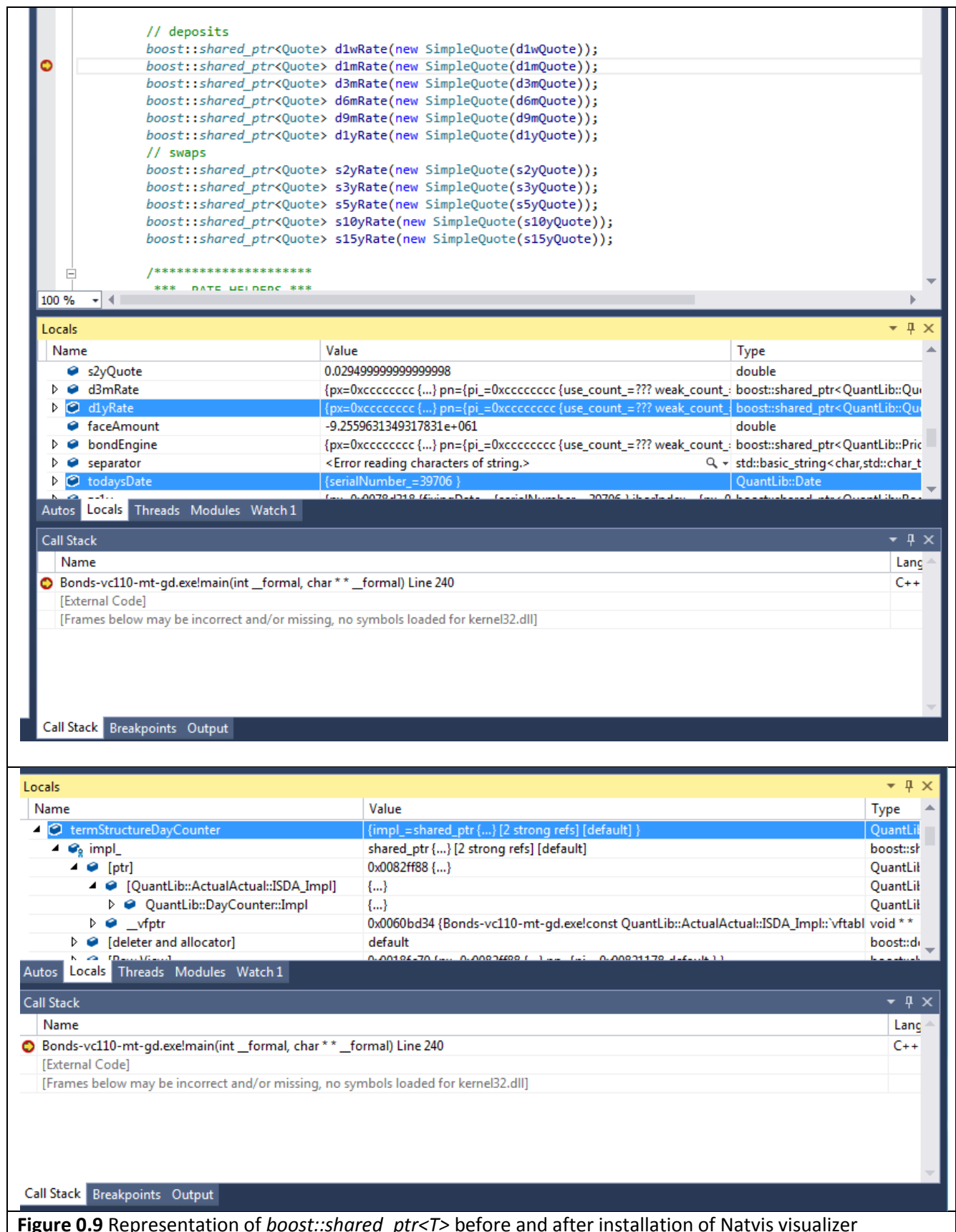


Figure 0.9 Representation of `boost::shared_ptr<T>` before and after installation of Natvis visualizer

Chapter 2: Recalling object oriented programming and getting started with date arithmetic.

Although this book assumes the basic knowledge of C++ (and thus OOP) we will at first quickly review its concepts. My experience shows that the students usually understand the main OOP principles but do not fully realize their advantages and their proper usage. The object oriented programming, as its name implies, works with *objects*. In turn, the objects are the instances of *classes*. Let us start with the class *Date* that is defined in `ql/time/date.hpp`

In order to open `date.hpp` quickly go to VASSISTX -> Open File in Solution and type `date`

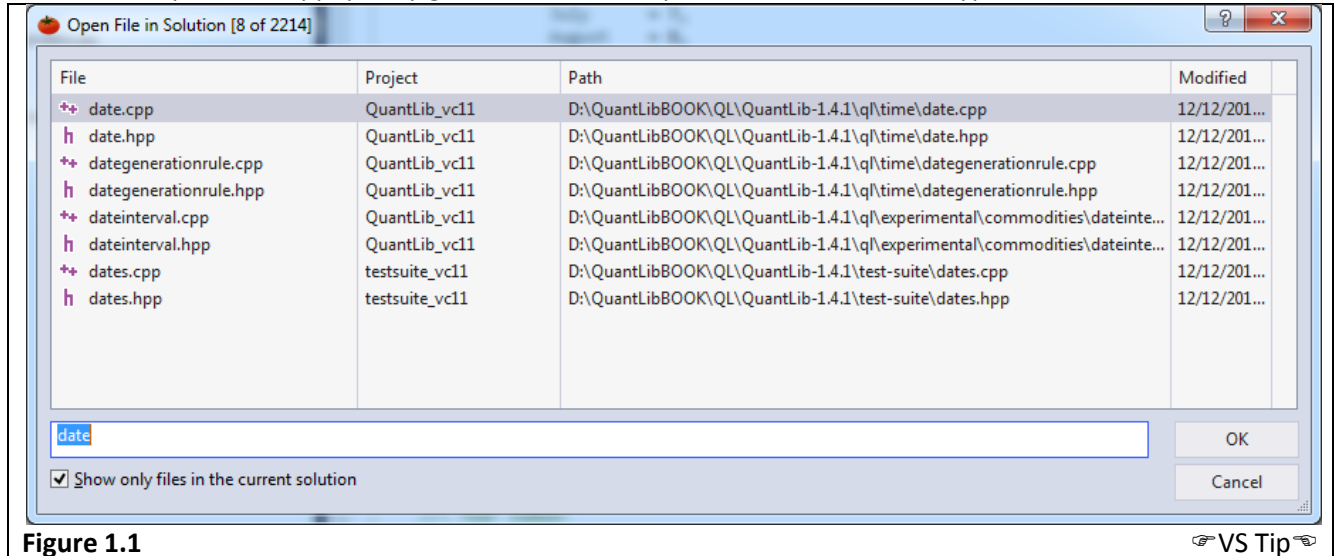


Figure 1.1

VS Tip

As every class, *Date* contains fields (a.k.a. attributes or member variables) and methods (the procedures and functions defined within the class). There is only one field: *serialNumber_* that contains the number of days counting from 01.12.1900. The underline at the end of the field's name is a coding convention, which lets us quickly distinguish the class member variables from the local variables in class methods²².

In accordance with the *encapsulation* (the 1st OOP principle) *serialNumber_* is *private*, which means that only the methods of the *Date* class²³ can access this field. There are also some private methods, for example *static void checkSerialNumber(BigInteger serialNumber)*. This method, as its name implies, checks whether a serial number (which uniquely represents a date) is valid.

Public and private methods separate, so to say, the interface and the realization. In this sense the private methods not only say that you are not allowed to call them but first of all they tell you that you will never need to call them! Indeed, you will not need *checkSerialNumber(BigInteger serialNumber)* when you are calculating any financial instrument. But obviously you may need the methods *dayOfMonth()*, *month()*, *year()* that are declared public. Moreover, QuantLib core team can change the implementation of *checkSerialNumber(BigInteger serialNumber)* or even rename/replace this method without a fear that it may damage the software projects that engage QuantLib.

²² Probably a better convention would be an understrike at the beginning, i.e. `"_serialNumber"` instead of `"serialNumber_"`. Then the Visual Studio Debugger would group all member variables together. However, you should better follow the established convention, especially if you are going to contribute your code to QuantLib.

²³ Additionally, the [methods of] the *friend* classes can. But *Date* has no friend classes.

You may have also noted that some methods are declared as *static* and some not. The latter have to do with objects whereas the former are defined for the classes as such. For example, a method call like *Date.weekday()* does not make sense since *Date* is a class (and not an object of the class *Date*). Thus trying to call *Date.weekday()* one actually says "tell me the weekday of date" (which date?!). But the command "tell me the weekday of 10.12.1979"²⁴ is absolutely correct.

On the other hand some properties of *Date* are the same for *all* objects of the class *Date*. Thus we can deal with them at the class (not at the object) level. For example the methods *minDate()* and *maxDate()* return the minimal and maximal dates, which QuantLib can digest (they are, respectively, 01.01.1901 and 31.12.2199). These are obviously class-level attributes. On the other hand the method *isEndOfMonth(const Date& d)* is also declared as static but it is not obvious whether it really belongs to the class level. One may also have defined it as non-static, since it is the property of a concrete date, whether it is the end of month or not. Moreover, I had an occasion to maintain a proprietary library that calculated loans and the date class in this library had an attribute *endOfMonth_*. The problem was as following: if a credit starts, say on the 28th of February and the installments are paid monthly, when should the next installment be paid: on the 28th or on the 31th of March?! It actually depended on loan agreement (ultimo or not). At first glance the introduction of the attribute *endOfMonth_* in the *Date* class seems ugly and a better solution would be to introduce something like *ultimo_* attribute in the class *Loan*. However, there were so many types of loans and their hierarchy was so complicated that the solution with the *endOfMonth_* as *Date* attribute was probably the easiest from the practical point of view. In other words: a good software architecture is very important but in practice there are always trade-offs between architecture, performance, code brevity and (last but not list) programmers' skills and personal preferences.

So far let us stop with theory for a while and do something with QuantLib. First of all add a new project *DateToy* to the QuantLib Solution. In Solution Explorer click on Solution 'QuantLib_vc11' -> Add -> New Project.

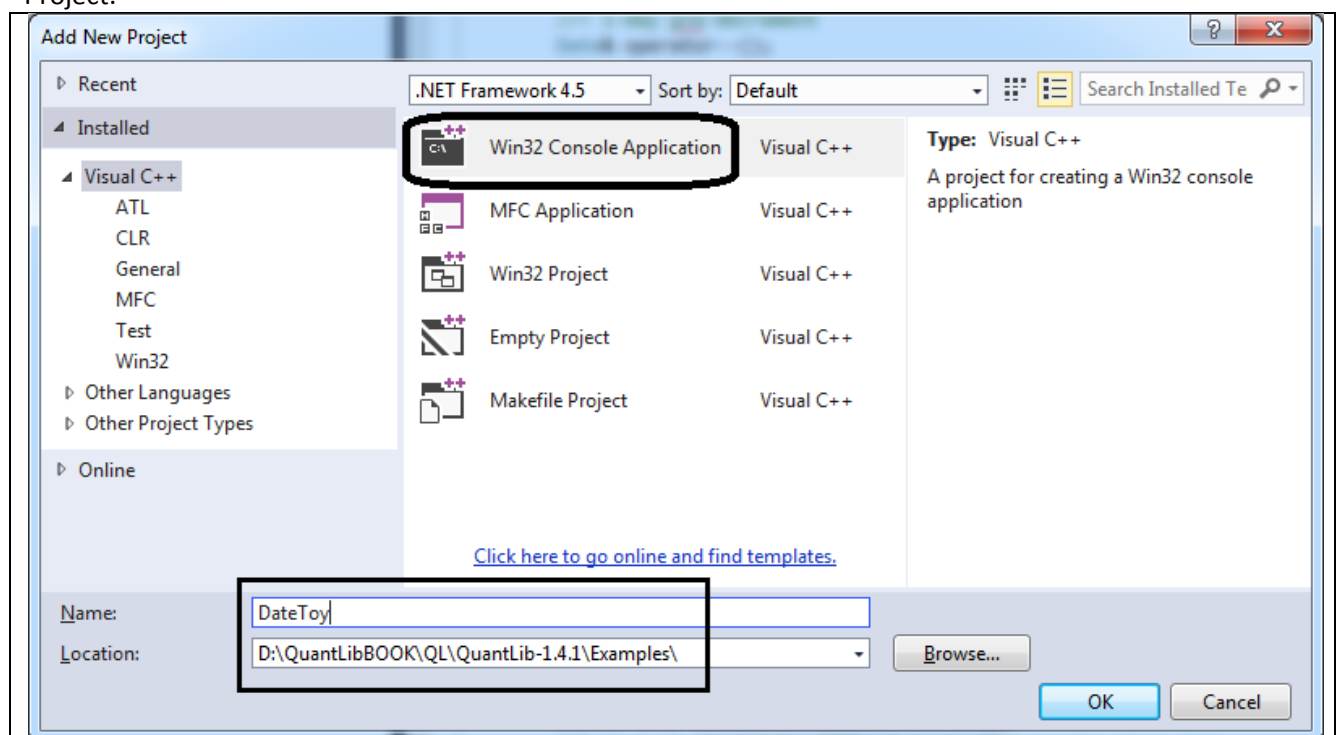


Figure 1.2 Adding new project to the QuantLib Solution

²⁴ In terms of C++/QuantLib: `(Date(10, December, 1979)).weekday();`
`QuantLib::Date(10, QuantLib::December, 1979)` instantiates the object of the class *Date* and *weekday()* returns the weekday of 10.12.1979 (it is Monday).

Select *Console application* and deactivate *Precompiled headers* and *Security Development Lifecycle (SDL) checks* (Figure 1.3).

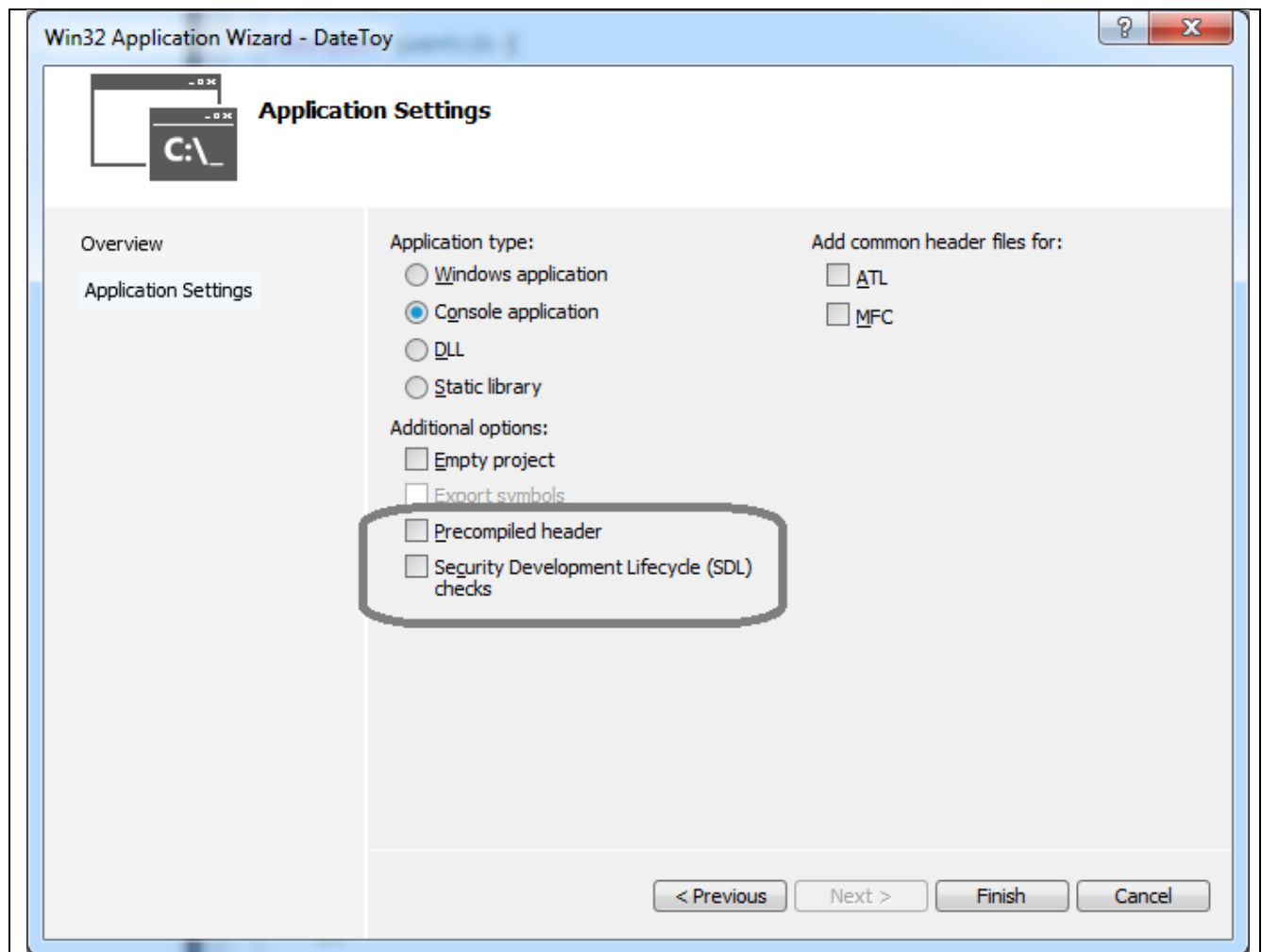


Figure 1.3 Application settings for the new project

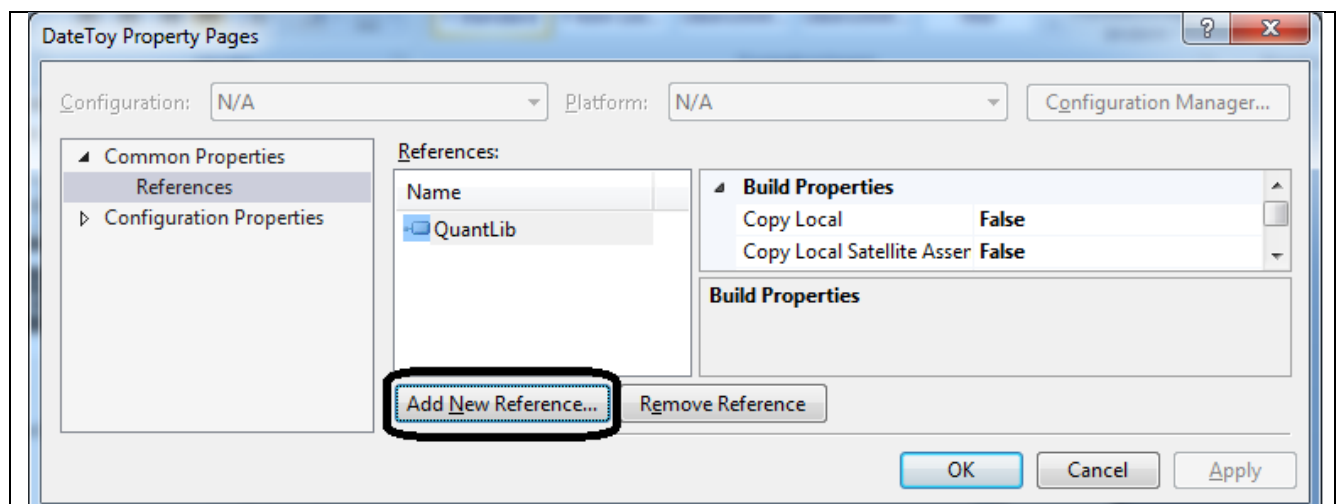


Figure 1.4 Add reference to QuantLib

Click Finish, a new project DateToy will appear in Solution Explorer. Click on this project, choose Properties -> Common Properties -> References and add reference to QuantLib (Figure 1.4).

Choose Properties -> Configuration Properties -> C++ and set Additional Include Directories to ..\..\%(AdditionalIncludeDirectories) (Figure 1.5).

Finally set the boost paths like we did before (Figure 0.4).

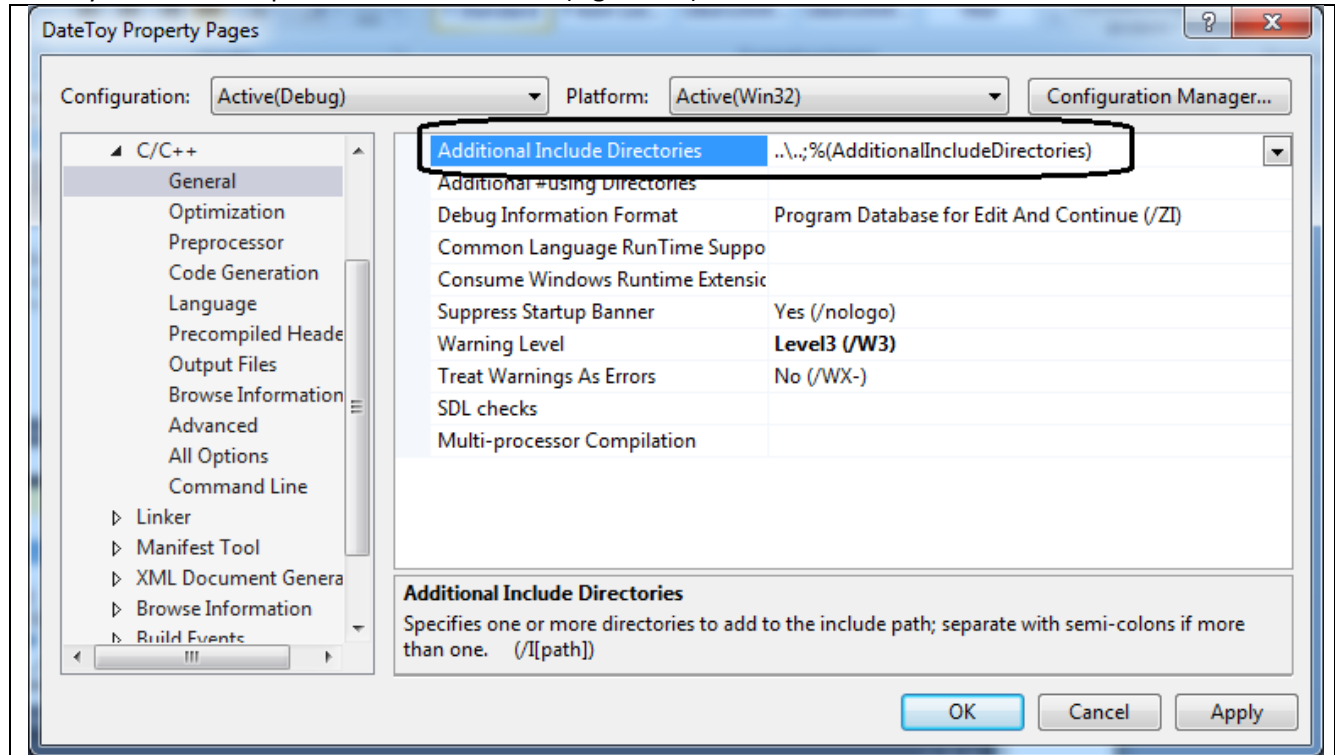


Figure 1.5 Additional Include Directories

```
#include <iostream>
#include <ql/time/date.hpp>

using namespace QuantLib;

int main()
{
    Date veryFirstDate(1, January, 1901);
    Date anotherVeryFirstDate(367);
    Date yetAnotherVeryFirstDate(veryFirstDate);

    std::cout << (Date(10, December, 1979)).weekday() << "\n";

    std::cout << veryFirstDate << "\n";
    std::cout << anotherVeryFirstDate << "\n";
    std::cout << yetAnotherVeryFirstDate << "\n";

    std::cout << Date::minDate() << "\n";
    std::cout << Date::maxDate() << "\n";

    return 0;
}
```

Code 1.1 Getting started with *Date* class

Now we call some methods of the *Date* class we discussed before. Open file *DateToy.cpp* and type Code 1.1. Set a breakpoint by "return 0;" line. Run the project. As you can see, the objects *veryFirstDate*, *anotherVeryFirstDate* and *yetAnotherVeryFirstDate* are essentially the same. But in order to instantiate them we have called three different constructors²⁵. In the first case we specified day, month, year and in the second case we explicitly specified the serialNumber. Finally, in the third case we engaged a so-called copy constructor (which, by the way, is not implemented in *Date* and thus automatically generated by C++ compiler)²⁶. Class *Date* also contains the definition of the *operator<<* or, more exactly, *std::ostream& operator<<(std::ostream& out, const Date& d)*. It allows to print the date on console by means of *std::cout <<*

Note that we write "::" to call the static methods.

Now let us add *veryFirstDate*, *anotherVeryFirstDate* and *yetAnotherVeryFirstDate* to the watch window. During debugging we can see (and even change) the properties of the watched objects. However, there is an annoying problem: since the dates are defined in terms of their serial numbers, we cannot really see much. Well, for the object *veryFirstDate* we have manually added the watches *veryFirstDate.year()*, *veryFirstDate.month()* and *veryFirstDate.dayOfMonth()* and thus we can see something. However, it is very cumbersome and each time we need manually refresh this watches since the Visual Studio does not update them automatically²⁷.

Watch 1		
Name	Value	Type
▶ <i>anotherVeryFirstDate</i>	{serialNumber_=367 }	QuantLib::Date
▲ <i>veryFirstDate</i>	{serialNumber_=367 }	QuantLib::Date
<i>serialNumber_</i>	367	long
▶ <i>yetAnotherVeryFirstDate</i>	{serialNumber_=367 }	QuantLib::Date
<i>veryFirstDate.year()</i>	1901	int
<i>veryFirstDate.month()</i>	January (1)	QuantLib::Month
<i>veryFirstDate.dayOfMonth()</i>	1	int
Autos Locals Threads Modules Watch 1		

Figure 1.6 Inspecting Date objects by debugging

In introduction we have mentioned *natvis* but in this case it would not help because its ability to call methods or run its own macros is very limited. But even if it would, we were still unable to flexibly *alter* the values of the member variables. In Appendix A we hack QuantLib and refactor the implementation of *Date*. This case is an excellent example of a *practical* refactoring: we are going to improve the program with little effort (relying on already existing business logic), have a nice idea how to do it ... and confront with the fact, that our idea is not completely chaste and requires some modification of the current business logic. However, if you are more interested in financial calculations than in programming aspects, you may skip Appendix A.

²⁵ Constructor is a special class method, which instantiates an object. Typically, a constructor validates input parameters (e.g. in our case day, month, year) and respectively assigns the class fields. A class can have many constructors (with different arguments).

²⁶ It is a very good practice to implement the copy constructor and assignment operator in your classes. If you don't C++ compiler will generate the default implementation (a bitwise copy of member variables). It may cause problems if e.g. your class contains pointers.

²⁷ Though it actually could since all these methods are *const*, i.e. they cannot change any class properties (except those that are declared as *mutable*).

Now let us finally do some financial calculations. Probably one of the most frequent (and simplest) is the calculation of a zero bond yield. However, even this simple calculation is not so easy. At first let us do the calculation and then analyze the code in detail.

```
#include <iostream>
#include <ql/quantlib.hpp>
#include <math.h>

using namespace QuantLib;

void main()
{
    //create a zero coupon bond
    Date issueDate(2, October, 2014);
    Natural settlementDays = 2;
    Calendar calendar = Germany(Germany::Settlement);
    BusinessDayConvention paymentConvention = Following;
    Real faceAmount = 100;
    Real redemption = 100;
    Date maturityDate(2, October, 2016);
    ZeroCouponBond zeroBond(settlementDays, calendar, faceAmount, maturityDate,
        paymentConvention, redemption, issueDate);

    //calculate its yield
    Real cleanPrice = 97.5;
    DayCounter dayCounter = ActualActual(ActualActual::ISDA);
    Compounding compounding = Compounded;
    Frequency frequency = Annual;
    Real accuracy = 1.0e-15;
    Size maxEvaluations = 10000;
    Rate yield = zeroBond.yield(cleanPrice, dayCounter, compounding,
        frequency, zeroBond.settlementDate(issueDate), accuracy, maxEvaluations);
    std::cout << "yield from QuantLib" << (double)yield << "\n";

    //and check the results manually
    double yearFraction2014 = (Date(31, December, 2014) - Date(7, October, 2014) + 1);
    yearFraction2014 /= 365;
    double yearFraction2016 = (Date(4, October, 2016) - Date(1, January, 2016));
    yearFraction2016 /= 366;
    double t = yearFraction2014 + 1.0 + yearFraction2016;

    double tDC = dayCounter.yearFraction(Date(7, October, 2014), Date(4, October, 2016));
    assert(abs(t - tDC) < accuracy);
    double yield2 = exp((Log(redemption) - Log(cleanPrice)) / t) - 1;
    std::cout << "yield from manual check" << yield2 << "\n";
    assert(abs((double)yield - yield2) < accuracy);
}
```

Code 1.2 Calculating yield of a zero bond

As you can see, in order to define a zero bond we need the following parameters:

- 1) *issueDate* is the date on which the bond is issued.

2) *settlementDays* are days that the backoffice needs to commit the transaction²⁸. In other words, if an investor buys a bond on *issueDate*, he will get it on (*issueDate* + *settlementDays*). In Germany two settlement days are common for the bonds. This rule is also known as T+2. Note that the interest starts accruing from (*issueDate* + *settlementDays*), not from the *issueDate*! The smallest time span for which the interest is accrued is one day.

Note that the variable "settlementDays" has the type "Natural". In Visual Studio, click on "Natural" and press Alt+F12 (Peek Definition). You will see in *types.hpp* the following definition:

```
typedef unsigned QL_INTEGER Natural; In turn, QL_INTEGER is nothing else but int. For QuantLib it is common to actively (ab)use the definition of new types. At first it really irritates but gradually a developer gets use to it and then it even becomes helpful.
```

3) *calendar* and *paymentConvention*. Normally, the interest accrues both for business days and holidays. But what if a settlement or redemption falls on a holiday (and how we detect it)?! The holidays are defined by *calendar*; *ql/time/calendars* contains the calendars for many countries. In our example we considered the basic German calendar (*Germany::Settlement*). Besides, there are special calendars for the most important German exchanges: *FrankfurtStockExchange*, *Xetra*, *Eurex*, *Euwax* (have a look at *germany.hpp*). The *paymentConvention* tells us what to do if a payment (or settlement) falls on a holiday. In our case the convention is "Following", which means that the date will be moved to the nearest following business day. In our example the *issueDate* (2.10.2014) is Thursday. Thus T+2, i.e. the settlement date (4.10.2014) falls on Saturday, which is not a business day. We need two business date for the settlement, so the settlement is, according to the "Following" rule is 7.10.2014 (Tuesday).

Moreover, the maturity date (2.10.2016) is also a non-business day (Sunday). According to the "Following" rule the bond should be redeemed on 3.10.2014 ... but it is a holiday according to our calendar²⁹ thus the maturity date is shifted to 4.10.2014.

4) *faceAmount* and *redemption* are self-explanatory. It is a pretty rare case that they are different but it is possible.

5) *maturityDate* is the date on which the redemption is paid. In case of a coupon bond the last coupon is usually also paid on maturity date. Note that the interest is usually accrued for the settlement day but not for the maturity day!

Further parameters we need are not the member variables³⁰ of the *ZeroCouponBond* class, yet we need them to compute the bond yield.

6) *cleanPrice* is (in case of a zero bond) just its market value. In case of a coupon bond one needs to distinguish between clean and dirty price due to partly accrued but not yet paid coupon. Clean price is more comfortable because it does not jump on the coupon payment dates.

²⁸ Though this is hardly relevant for the financial calculation I do recommend you to read about settlement risk (http://en.wikipedia.org/wiki/Settlement_risk) and delivery versus payment (http://en.wikipedia.org/wiki/Delivery_versus_payment)

²⁹ Tag der Deutschen Einheit (German Unity Day). Interestingly, it is a business day for *FrankfurtStockExchange*, *Xetra*, *Eurex*, *Euwax*... though as far as I remember there is no trading activity on these exchanges on the 3rd of October.

³⁰ Note that contrast to *ZeroCouponBond* the class *FixedRateBond* does have the member variable *dayCounter_* ... though it seems to be unused; at least neither doxygen call graph, nor Visual Assist "find reference", nor Visual Studio "call hierarchy" tells us where the method *FixedRateBond::dayCounter()* is called or where *dayCounter_* is directly addressed.

7) *dayCounter*, as its name implies, counts the number of [interest] days and the year fraction between two dates. On one hand we said that a day is the basic time span for which the interest accrues. On the other hand a year is the basic time unit for the interest rate arithmetic.

In our case we used `ActualActual::ISDA`. It works as follows: first of all it splits the time span by years: in our example to $[7.10.2014, 31.12.2014] \cup [1.01.2015, 31.12.2015] \cup [1.01.2016, 4.10.2016]$. There are 86 days in $[7.10.2014, 31.12.2014]$ and 2014 is not a leap year, so we divide 86 by 365 to get the year fraction. The year 2015 is completely within the time span, so it gives the year fraction equal to 1.0 (and it does not matter whether it a leap year or not). Finally, there are 277 interest³¹ days in $[1.01.2016, 4.10.2016]$ and 2016 is a leap year. So we divide 277 by 366 and the whole time span is equal to $86/365 + 1.0 + 277/366 = 1.9924$ years.

There are pretty many day counters, have a look at `ql/time/daycounters`. Their logic is pretty straightforward and you can readily understand it from the source code and comments.

But there is one nuance: in case of the `ActualActual::ISMA` the year fraction additionally depends on the reference period! That's why the method *yearFraction* in the basic abstract class *DayCounter* is declared as follows:

```
//! Returns the period between two dates as a fraction of year.
Time yearFraction(const Date&, const Date&,
                  const Date& refPeriodStart = Date(),
                  const Date& refPeriodEnd = Date()) const;
```

8) *compounding* means the interest rate compounding rule. QuantLib supports *Simple* $(1 + rt)$, *Compounded* $(1 + r)^t$, *Continuous* $\exp(rt)$ and *SimpleThenCompounded* rules. *Compounded* is the most common rule in Germany.

9) *frequency* means something periodical, e.g. how often the coupons are paid. A zero bond has no coupons but still the frequency may be relevant because by *Compounded* rule we actually have³² $(1 + \frac{r}{\text{freq}})^{t \cdot \text{freq}}$

Nice to know, since it is not mentioned in `ql/compounding.hpp`!

10) *accuracy* and *maxEvaluations* determine the numeral precision of the bond yield. In Germany it is required to calculate with $1.0e-6$ precision but since in our case we check the yield by means of a closed form solution, we set very high precision to check whether the numerical iteration method can achieve it (it can).

After we calculated the bond yield with QuantLib business logic, we check it manually. Note that we add one day in year 2014 and do not do it in year 2016 because the settlement day is an interest day but the maturity day is not.

Also note the `assert(abs(t - tDC) < accuracy)` statement, it is very practical to use it if you expect that a certain condition in your program always takes place.

You might be quite confused with what you have just learnt: compounding rules, day counters, frequencies... At least I was very confused as I confronted them - and not due to the fact that the reality is more complicated than $\exp(rt)$ but rather because there are so many of them. But do not panic! First of all, you will hardly need to learn all of them in detail. Actually, they are already implemented in QuantLib and you just need to understand how to use them properly. And you can always have a look at <http://www.opengamma.com/sites/default/files/interest-rate-instruments-and-market-conventions.pdf>

³¹ Recall that the last day (i.e. maturity day) is not an interest day.

³² Lines 53-54 from `ql/interestrate.cpp`

case Compounded: `return std::pow(1.0+r_/freq_, freq_*t);`

This excellent reference "contains what everybody is supposed to know when they first start working in the industry".

Our case with a zero bond is not only an excellent opportunity to overview the realm of the market conventions. It is also good for discussing the [limitations of] second basic OOP principle, the inheritance. In some introductory books on OOP the inheritance is interpreted as possibility to evolve from simple to complex, starting with a very basic [abstract] class and making derived classes functionality richer and richer. Some methods are declared but not implemented in basic classes and it is the task of the derived classes to have them implemented. If necessary, the derived class may also override the implementation of the basic class(es) methods. In a sense we can see it at Figure 1.7. The basic "financial" class is the abstract class *Instrument*. This class is not the first in class hierarchy, it inherits from *LazyObject*, which in turn inherits from *Observer* and *Observable*.

LazyObject is a *design pattern* with motto "(re)calculate only when you need it". The goal is the optimization of the computational performance (e.g. it may be very costly to recalculate 100000 Monte Carlo paths). *Observer-Observable* design pattern is as follows: when an *Observer* changes its state, it notifies all its *Observables*; in this case the *Observables* (that are often *LazyObjects*) usually need to recalculate themselves.

The *Observer*, *Observable* and the *LazyObject* classes implement purely technical features, which (according to the philosophy of QuantLib) each financial instrument should possess in order to make financial computations faster and more reliable from the technical point of view. As we have already mentioned, "financial hierarchy" begins with *Instrument*. It has some very basic properties (and respective methods) that each financial instrument should have: e.g. every (traded) instrument should obviously have an NPV (net present value). But e.g. the calculation of a bond NPV is very different from the calculation of an option NPV (and yet both are financial instruments). Thus *Instrument* class "is purely abstract"³³ and defines the interface of concrete instruments which will be derived from this one"³⁴.

Respectively, *Bond* inherits all methods from *Instrument* (and implements some of them). Additionally, it has some bond-only specific methods. So far we can see a perfect realization of the "from simple to complex" paradigm. However, there are some problem in this sense with *ZeroCouponBond*. On the one hand a *ZeroCouponBond* is a special case of a [coupon] *Bond* (which is fully in accordance with class hierarchy that we see at Figure 1.7). On the other hand the functionality of a *ZeroCouponBond* is much simple than that of a [coupon] *Bond*.

What does it all mean? Just the fact, that there is not perfect programming paradigms that perfectly cover all situations in real life. But it also means that inheritance paradigm is sometimes misunderstood. "From simple to complex" is not the only goal of inheritance. Actually, some experts recommend to replace inheritance with encapsulation because long inheritance paths makes program less readable (e.g. have a look at the Doxygen class diagram for *CallableZeroCouponBond*). Additionally, it becomes more and more difficult to maintain logically consistent class hierarchy, e.g. why³⁵ both *FixedRateBond* and *ZeroCouponBond* are derived directly from *Bond* but *CallableZeroCouponBond* is derived from *CallableFixedRateBond*, which in turn is derived from *CallableBond*.

³³ `//! Abstract instrument class`

`/*! This class is purely abstract and defines the interface of concrete instruments which will be derived from this one.`
Strictly speaking, from the technical point of view this class is not purely abstract since some methods, e.g. `setPricingEngine()` are implemented. But from the business logic point of view it may be called purely abstract.

³⁴ Cited from the class description in the source code of `Instrument.hpp`

³⁵ Probably because *CallableZeroCouponBond* inherits the overridden method `setupArguments()` from *CallableFixedRateBond*. But this is a weak argument since one could have implemented this methods directly in *CallableBond* because *CallableFixedRateBond* is the only class, which inherits from *CallableBond*.

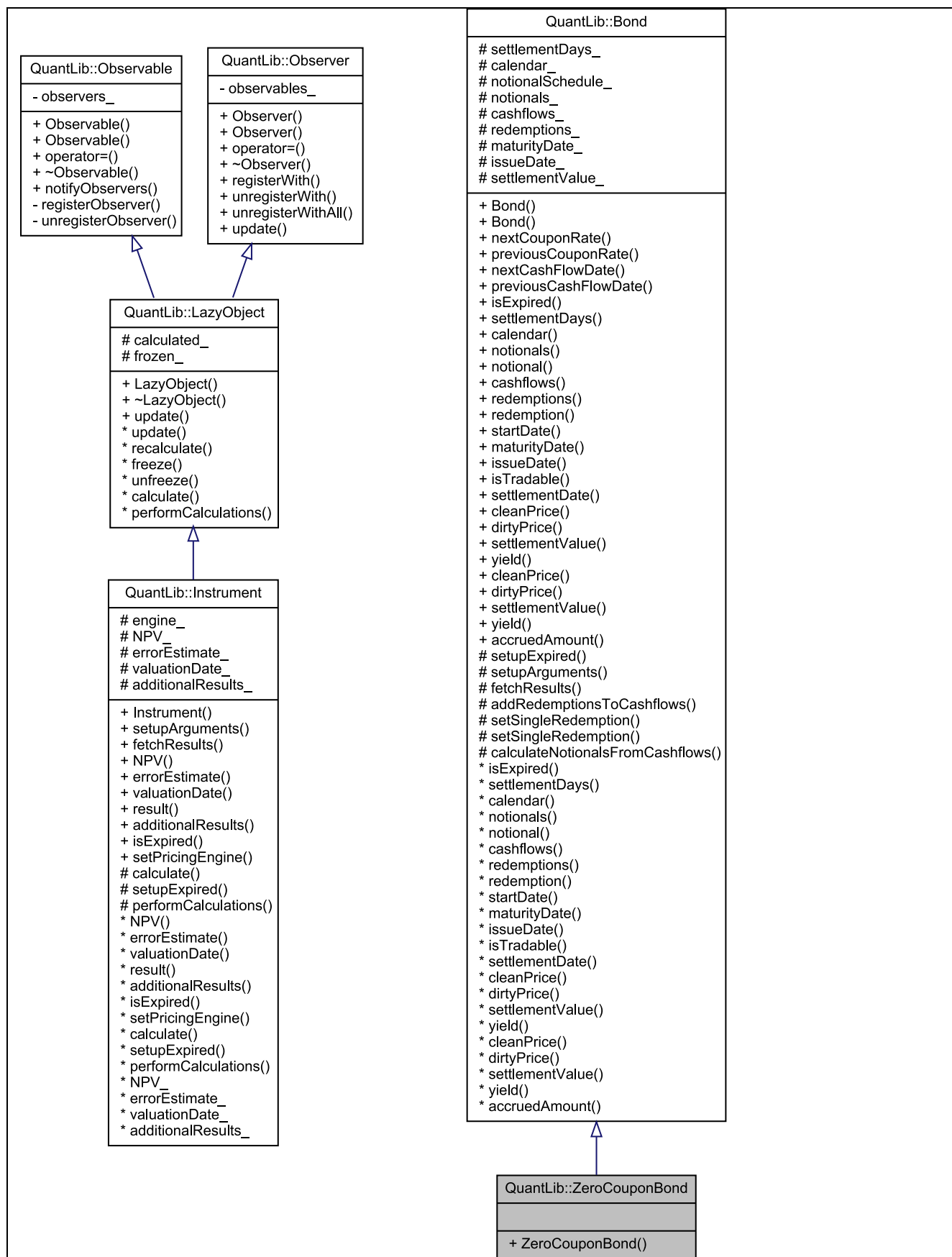


Figure 1.7 ZeroCouponBond in QuantLib class hierarchy (Doxygen-generated diagram)

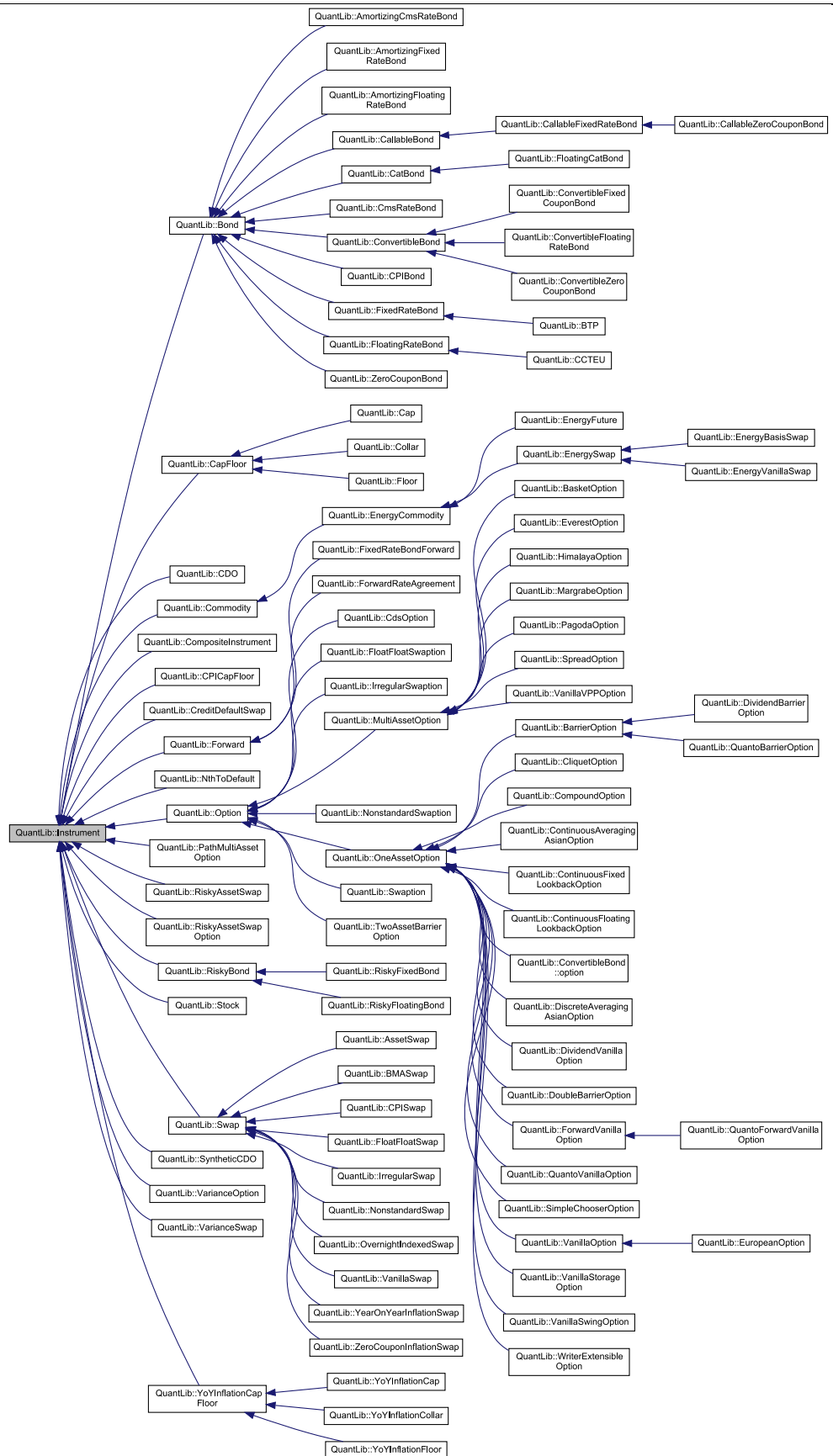


Figure 1.8 The realm of classes that inherit from *QuantLib::Instrument*

But inheritance (together with the 3rd OOP principle polymorphism) has another important goal (which often remains unexplained in introductory books): programmers must be able to reduce the objects to the properties and methods of the [very] basic classes. Figure 1.8 shows how many classes inherit from *Instrument*. Now imagine that we need to build a risk modeling engine on the basis of QuantLib in order to simulate the evolution of our portfolio. In this context it does not matter which instruments we have in our portfolio. All we need to model is the evolution of NPV. Thanks to polymorphism we need not to distinguish between the dozens of classes but can just call the NPV related methods that are declared in *Instrument*. A compiler will by itself call the proper implementations of these methods (that may be different in derived classes). But in order to do this a compiler needs an inheritance hierarchy³⁶.

Finally, in order to complete the discussion of the [basics of] date arithmetic, I highly recommend you to have a look at the blog post "Odds and ends: date calculations" by Luigi Ballabio:

<http://implementingquantlib.blogspot.de/2014/11/odds-and-ends-date-calculations.html>

In particular, this post discusses *Periods* and *Schedules*.

Periods are not as straightforward as they may seem to be at first glance but the nuances are mostly technical. In either case, at least at first approximation Periods may be comprehended as counterparts of Frequencies (which we have already encountered). In particular, the Periods are used in such a way to construct Schedules. Luigi writes in his blog post, "Schedule class ... is used to generate sequences of coupon dates". This is definitely the most frequent but not the only usage of *Schedule*. Have a look at Doxygen caller graphs for the methods of *Schedule* and you will see that it is used by *MarketModel*, *CDO*, *CPISwap*.

Schedule is thoroughly explained by Dimitri Reiswich: <http://quantlib.org/slides/dima-ql-intro-1.pdf> (Slide 32). Let us construct a schedule and then use it to create a coupon bond.

The section "bond parameters" is already familiar to us from the Code 1.2. There is a new parameter *accrualConvention*, which is set to *Unadjusted*. A schedule will generate a sequence of the coupon periods. But if the begin or the end of a such period falls on a holiday, shall we shift it (somehow) to a working day? At the first glance we shall not because, as we have already mentioned, all calendar days are interest days. On the other hand the payment will be shifted (since one can hardly commit a financial transaction on a holyday). In our example the end of the last coupon period is 02.10.2016 (Sunday) and 03.10.2016 is a holiday, so this coupon will be paid on 04.10.2016. But this means that no interest will be accrued for 02.10.2016 and 03.10.2016. Setting *accrualConvention* equal to *paymentConvention* would fix this problem ... but there is no 100% ubiquities convention how to relate coupon periods and coupon payments. That's the business day conventions are set separately for both of them.

The next section makes to schedules: making *schedule* is straightforward. Note that it is a common practice (at least in Germany) to generate payments backwards from the end date. Also note that that the Period is defined in terms of frequency.

In either case, the constructor has a lot of parameters and it might be not so easy to memorize all of them (at least it is not clear for what *accrualConvention* stands for two times). To solve this problem we may apply the Named Parameter Idiom. Technically it is implemented as follows: calling *MakeSchedule()* creates an object of class *MakeSchedule*. All other methods we call set a respective parameter and return the reference to the same object. For example the method *from* is defined as

³⁶ Alternative approach in C++ is the template programming, which does not require a shared ancestor class. The question whether to choose inheritance or templates is pretty usually pretty straightforward: if there is an "obvious" class hierarchy then choose the former otherwise choose the latter. But in general you should never mix them! In chapter 3 we consider the class *PiecewiseYieldCurve* and encounter such a mixture, which is pretty hard to understand.


```

    MakeSchedule& MakeSchedule::from(const Date& effectiveDate) {
        effectiveDate_ = effectiveDate;
        return *this;
    }

```

Note "&" in `MakeSchedule&` i.e. it returns a reference and also note that there is so far an object of `MakeSchedule`, not of `Schedule` class!

Finally, as the assignment operator (`MakeSchedule::operator Schedule()`) is called, it creates an object of class `Schedule`.

Then we just check that `schedule` and `schedule2` are identical.

```

#include <iostream>
#include <ql/quantlib.hpp>
#include <math.h>

using namespace QuantLib;

void main()
{
    //bond parameters
    Date issueDate(2, October, 2014);
    Natural settlementDays = 2;
    Calendar calendar = Germany(Germany::Settlement);
    BusinessDayConvention paymentConvention = Following;
    BusinessDayConvention accrualConvention = Unadjusted;
    Real faceAmount = 100;
    Real redemption = 100;
    Date maturityDate(2, October, 2016);
    DayCounter dayCounter = ActualActual(ActualActual::ISDA);
    Frequency frequency = Annual;

    //make schedules via constructor
    Schedule schedule(issueDate, maturityDate, Period(frequency), calendar,
        accrualConvention, accrualConvention, DateGeneration::Backward, false);

    //... and via Named Parameter Idiom helper
    Schedule schedule2 = MakeSchedule().from(issueDate).to(maturityDate).
        withFrequency(frequency).withCalendar(calendar).
        withConvention(accrualConvention).
        withTerminationDateConvention(accrualConvention).
        backwards().endOfMonth(false);

    //check that the schedules are identical
    std::vector<Date> datesSchedule = schedule.dates();
    std::vector<Date> datesSchedule2 = schedule2.dates();
    assert(datesSchedule.size() == datesSchedule2.size());
    for (unsigned int d = 0; d < datesSchedule.size(); d++)
    {
        assert(datesSchedule[d] == datesSchedule2[d]);
        std::cout << "Schedule Date " << d + 1 << " " << datesSchedule[d] << "\n";
    }

    //coupons
    std::vector<Rate> coupons(datesSchedule.size() - 1, 0.05);

    //bonds
    FixedRateBond couponBond(settlementDays, faceAmount, schedule, coupons,
        dayCounter, paymentConvention, redemption, issueDate, calendar);
}

```

```

FixedRateBond couponBond2(settlementDays, calendar, faceAmount, issueDate,
    maturityDate, Period(frequency), coupons, dayCounter, accrualConvention,
    paymentConvention, redemption, issueDate, Date(), DateGeneration::Backward,
    false, calendar);

//bond cashflow
Leg bondCashflow = couponBond.cashflows();
for (unsigned int d = 0; d < bondCashflow.size(); d++)
    std::cout << "Payment " << d + 1 << ": " << (bondCashflow[d])->date()
        << " " << (bondCashflow[d])->amount() << "\n";

//check that both bonds pay the same interest
for (Date d = issueDate; d <= maturityDate; d++)
{
    double accruedInterest = couponBond.accruedAmount(d);
    double accruedInterest2 = couponBond2.accruedAmount(d);
    std::cout << "Interest accrued on " << d << " " << accruedInterest << "\n";
    assert(abs(accruedInterest - accruedInterest2) < 1.0e-15);
}
}

```

Code 1.3 Constructing schedules and coupon bonds

Further we generate two coupon bonds, one with `schedule` we constructed before and one via constructor (which internally generates a schedule). Obviously, you should favor the former, since the process is split in two steps and you can check the schedule before creating a bond. As we have already noted, coupon payment dates and the end of coupon periods may differ.

Finally, we check that the interest is accrued identically by `couponBond` and `couponBond2`. There is a curiosity³⁷: in the constructor of the *Bond* class (which is parent for *FixedRateBond*) there is a following plausi-check:

```

if (issueDate_ != Date()) {
    QL_REQUIRE(issueDate_ < cashflows_[0]->date(),
        "issue date (" << issueDate_ <<
        ") must be earlier than first payment date (" <<
        cashflows_[0]->date() << ")");
}

```

I would expect something like

```

QL_REQUIRE(
    calendar_.advance(issueDate_, settlementDays_, paymentConvention_) < cashflows_[0]->date()
)

```

i.e. that no payment are allowed earlier than settlement date. But it is not the case, moreover, you cannot easily hack the QuantLib and modify this plausi-check in such a way because `paymentConvention_` is not a member of *Bond* (and not even of *FixedRatebond*). Rather it is a member of `Leg cashflows_` (which, in turn, is a member of *Bond* but is initialized in the constructor of *FixedRateBond* after the call of the parent constructor; this it is impossible to get `paymentConvention_` within the constructor of *Bond* even if `cashflows_` would store the `paymentConvention_` as member variable (it does not)).

³⁷ Skip it if is too hard to comprehend, it is just technical details that were curios for *me*.

Chapter 3: (Yield) Term Structures

From the programmer's point of view the term structures are well-documented in Luigi Ballabio's book: <https://dl.dropboxusercontent.com/u/13584583/qlbook/chapter3.pdf>, which you are certainly recommended to have a look at. The most important term structures are the yield term structures. As Luigi himself states "it was even called TermStructure back in the day, when it was the only kind of term structure in the library and we still hadn't seen the world". In this chapter we will discuss the yield term structures by means of two practical examples.

Example 1: As I was a recent graduate I actively looked for job and once got the following exercise:

Market Data 1 (maturity in years, annual coupon, annually compounded rates, face value 100)											
	Bond1	Bond2	Bond3	Bond4	Bond5	Bond6	Bond7	Bond8	Bond9	Bond10	Bond11
Maturity	1	2	3	4	5	6	7	8	9	10	9
Coupon	4	5	5	4	5	6	6	6	6	5	3
PV	101.5	103	102.5	98	101	105	104.5	104	104	96	???

Market Data 2 (maturity in years, annual coupon, annually compounded rates, face value 100)											
	Bond1	Bond2	Bond3	Bond4	Bond5	Bond6	Bond7	Bond8	Bond9	Bond10	Bond11
Maturity	1.7	2.1	3	3.8	4.6	5.6	6.5	7.2	8	9	5.4
Coupon	4	5	4	5	4	6	4	5	4	5	4
PV	104.2	110	103	105	100	103	96	103	92	98	???

Zero rate from 0 to 1 year is assumed constant and equal to 0.02

Question: What is the present value of bond 11 using a bootstrapping algorithm?

The maturities $m(i)$ of bonds 1 to 10 may be assumed to be increasing and chosen so that $m(i+1)-m(i) \leq 1$. Please use annually compounded rates. Both Excel-based and VBA-based solutions are accepted.

The first part is pretty straightforward, such exercises are done by virtually any MFE Program. First of all we find the yield of the first bond r_1 . It holds

$$\frac{\text{coupon}_1 + \text{FaceValue}_1}{1+r_1} = PV_1 \text{ i.e. } \frac{100+4}{1+r_1} = 101.5 \text{ so } r_1 = 0.024630$$

Further we have

$$\frac{\text{coupon}_2}{1+r_1} + \frac{\text{coupon}_2 + \text{FaceValue}_2}{(1+r_2)^2} = PV_2 \text{ i.e. } \frac{5}{1.024630} + \frac{100+5}{(1+r_2)^2} = 103.0$$

Thus

$$(1+r_2)^2 = 105/98.120190 = 1.070116$$

and respectively

$$r_2 = \exp(\ln(1.070116)/2) - 1 = 0.034464$$

It is the so-called *bootstrapping* algorithm, which can be continued iteratively. Of course we need not solve it manually, rather we can solve it like in Code 3.1, for which we even do not need QuantLib. The only problem is that it is never the case in real world.

The second case is much more realistic. However, there is no non-ambiguous algorithm to solve it. Obviously, we need to interpolate but what - yields, discount factors or something else? And of course there is no the only right way to connect the dots. As I tried to solve this problem I tried several plausible approaches but none of them has satisfied my interviewer. Then he gave me a hint to "use linear

interpolation" (of what?!) ... I did my best but he still was not satisfied. I think I still would not satisfy this guy if I had solved the problem with QuantLib ... but probably I could have impressed him with the knowledge of common term structure fitting approaches (that are rarely learnt in detail by MFE programs).

```
#include <iostream>
#include <math.h>

void main()
{
    //market data
    double coupons[] = { 4.0, 5.0, 5.0, 4.0, 5.0, 6.0, 6.0, 6.0, 6.0, 5.0 };
    double pvs[] = { 101.5, 103.0, 102.5, 98.0, 101.0, 105.0, 104.5, 104.0, 104.0, 96.0 };
    const double FACE_VALUE = 100.0;

    //iteratively calculate yields by bootstrapping algorithm
    const int n = sizeof(coupons) / sizeof(coupons[0]);
    double yields[n];
    yields[0] = (FACE_VALUE + coupons[0]) / pvs[0] - 1;
    std::cout << "yield 1: " << yields[0] << "\n";
    for (int i = 1; i < n; i++)
    {
        double couponsPV = 0.0;
        for (int c = 1; c <= i; c++)
        {
            couponsPV += coupons[c] / pow((1.0 + yields[c - 1]), c);
        }
        yields[i] = exp(log((FACE_VALUE + coupons[i]) / (pvs[i] - couponsPV)) / (i + 1)) - 1;
        std::cout << "yield " << i + 1 << ": " << yields[i] << "\n";
    }

    //answer the question
    const double bond11Coupon = 3.0;
    const int bond11Maturity = 9;
    double bond11PV = 0.0;
    for (int i = 1; i <= bond11Maturity; i++)
    {
        bond11PV += bond11Coupon / pow((1.0 + yields[i - 1]), i);
    }
    bond11PV += FACE_VALUE / pow((1.0 + yields[bond11Maturity - 1]), bond11Maturity);
    std::cout << "PV of the Bond11 is " << bond11PV << "\n";
}
```

Code 3.1 Bootstrapping algorithm in an ideal case

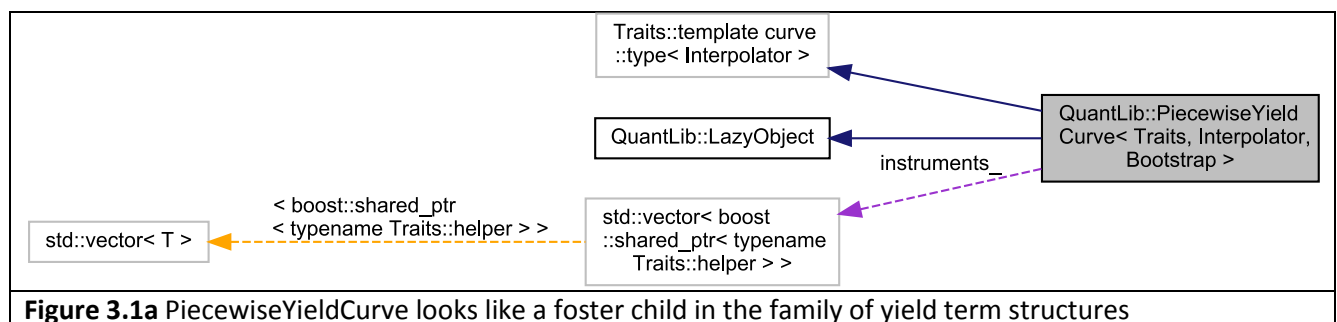
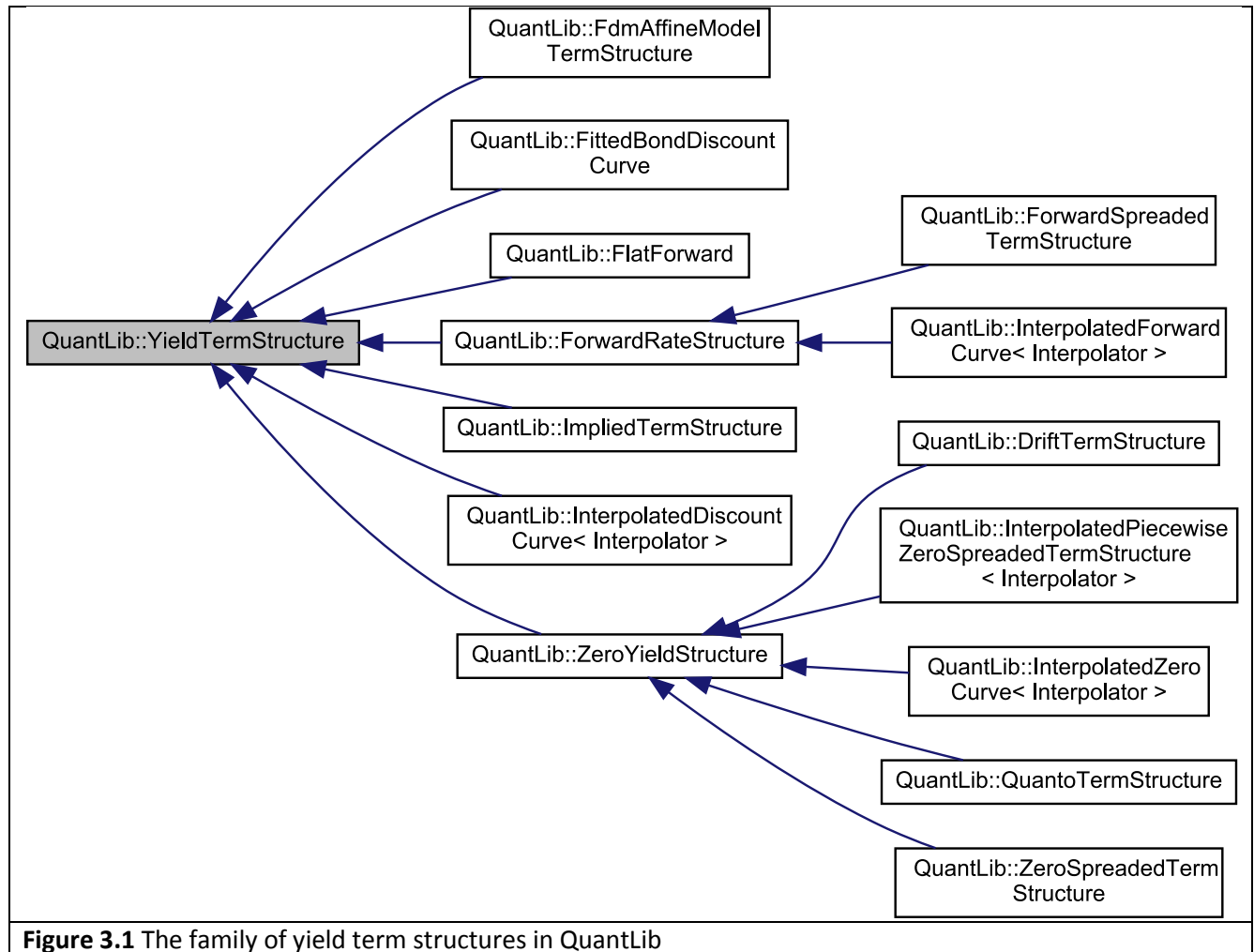
Let us have a look at Figure 3.1 to find a suitable tool for our task. At first glance it seems that we need to take *FittedBondDiscountCurve*³⁸. There is a small nuance with *FittedBondDiscountCurve*, it contains a class *FittingMethod*, which is a base class for several fitting methods, as Figure 3.2 shows. But this nuance is not a problem, the specification of the *FittingMethod* is straightforward and Google readily helps to grasp the ideas behind the fitting methods.

But there is also a big nuance: Figure 3.1 does not give us the full picture! There is class *QuantLib::PiecewiseYieldCurve*, which allows us to adhere to the hint to "use linear interpolation"³⁹, whereas

³⁸ Note that though we can easily calculate the yields of our bonds, the *ZeroYieldStructure* will not do because we have coupon bonds, not zero bonds.

³⁹ An advanced quant developer (like me ☺) often confronts the following problem: he sees that the software requirements he needs to implement are actually suboptimal. But unauthorized deviations from the requirements are

FittedBondDiscountCurve does not allow it. But it seems that *PiecewiseYieldCurve* does not inherit from *YieldTermStructure*! Indeed it does, but in a pretty complicated manner. As we can see from Figure 3.1a, *PiecewiseYieldCurve* intensively (ab)uses templates. Luigi explains this complex construction in his book in detail. Unless you are really curious, you do not need to spend your time on them. Just to put it briefly: *PiecewiseYieldCurve* inherits from *Traits::template curve<Interpolator>::type*, which is a template class. Meaningful Traits are defined in `/ql/termstructures/yield/bootstraptraits.hpp`, there are (currently) three structures: *Discount*, *ZeroYield* and *ForwardRate*.



usually unwelcome. So in this case the best solution would be to implement both the required approach and a better alternative (explaining, why the latter is better).

In turn, each Trait-structure contains a struct curve, e.g. in case of *Discount* it is

```
struct curve {
    typedef InterpolatedDiscountCurve<Interpolator> type;
};
```

and *InterpolatedDiscountCurve<Interpolator>* does inherit from *YieldTermStructure* (see Figure 3.1).

Thus *PiecewiseYildCurve* inherits from *YieldTermStructure* via Traits. But *Traits* is a template class and as long as it is unspecified, the inheritance cannot be recognized.

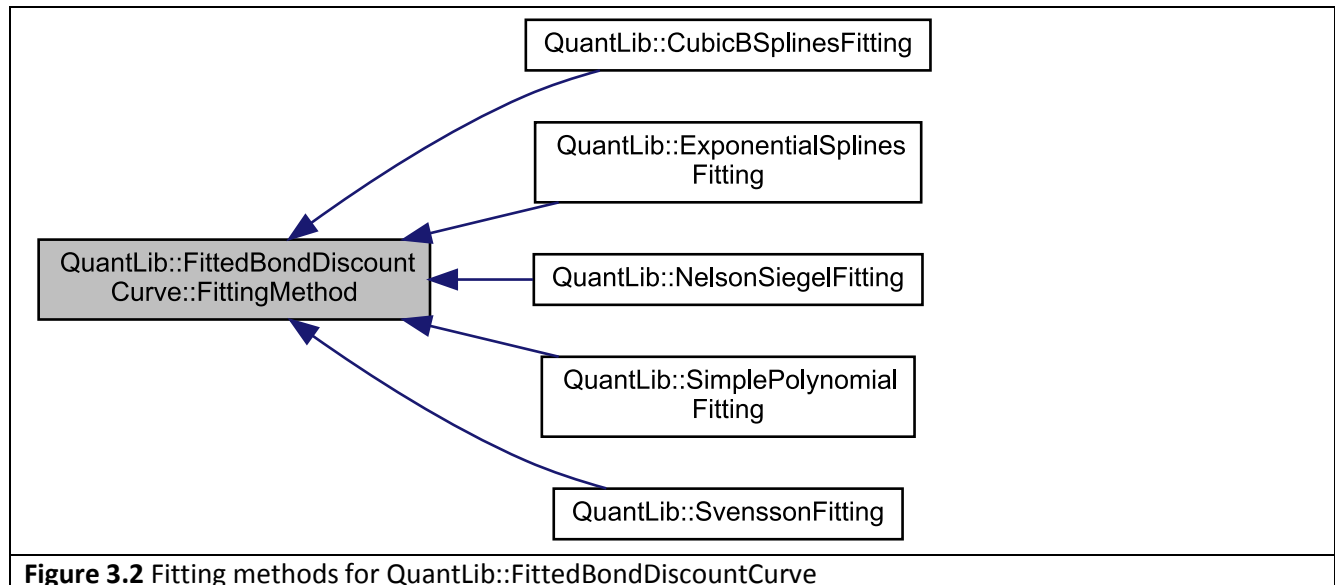


Figure 3.2 Fitting methods for QuantLib::FittedBondDiscountCurve

Code 3.2 is in a sense a slightly adopted (but thoroughly explained) version of the code from **FittedBondCurve** (a sample project from the QuantLib suite).

Code 3.2 brings us a lot of interesting features. First of all, we should clarify whether the bond prices are clean or dirty⁴⁰. Since the former is usually the case, we assume it (under this assumption it is also somewhat easier to work with QuantLib). Further do not be confused with `numberOfBonds = 11`. The first bond (with coupon of 0.02 and maturity of 1.0) just depicts the statement "Zero rate from 0 to 1 year is assumed constant and equal to 0.02".

Further we confront something new: `std::vector< boost::shared_ptr<SimpleQuote> > quote; .`

SimpleQuote (which inherits from the abstract basis class *Quote*) is just a structure to keep (whatever) market quotes. It contains a double value and a bool flag whether a quote is still valid (you know, the market quotes: prices, implied volatilities, interest rates and so on change quickly ☺) .

As to `boost::shared_ptr<T>`, it is a shared pointer⁴¹ (a member of smart pointers family), a very practical programming trick. Actually, it automates the memory management (to some extent it resembles Java references). The implementation of the shared pointers in boost is complicated but essentially its member variables are `T *px` (a conventional pointer) and the reference counter `boost::detail::shared_count pn`. Every time a shared pointer is passed as an argument, the reference counter is increased by one. The data, to which `px` points to is allocated in heap (since `px` is just a normal pointer).

⁴⁰ In previous task it was not an issue since there were no fractional maturities and thus no fractional coupons.

⁴¹ Dimitri Reiswich provides a concise hands-on introduction to boost shared pointers:

<http://quantlib.org/slides/dima-boost-intro.pdf>

Also recall that we setup Natvis visualizer for boost by Arkady Shapkin, in particular, they contain visualizers for shared pointers.

```

#include <iostream>
#include <ql/quantlib.hpp>
#include <math.h>

using namespace QuantLib;

#define LENGTH(a) (sizeof(a)/sizeof(a[0]))

void main()
{
    const Size numberOfBonds = 11;
    Real cleanPrices[] = { 100.0, 104.2, 110, 103, 105, 100, 103, 96, 103, 92, 98};
    Real lengths[] = { 1.0, 1.7, 2.1, 3, 3.8, 4.6, 5.6, 6.5, 7.2, 8, 9 };
    Real coupons[] = { 0.02, 0.04, 0.05, 0.04, 0.05, 0.04, 0.05, 0.04, 0.05, 0.04, 0.05 };
    const double FACE_VALUE = 100.0;
    const double COUPON = 4.0;
    const double MATURITY = 5.4;

    std::vector< boost::shared_ptr<SimpleQuote> > quote;
    for (Size i = 0; i < numberOfBonds; i++) {
        boost::shared_ptr<SimpleQuote> cp(new SimpleQuote(cleanPrices[i]));
        quote.push_back(cp);
    }

    RelinkableHandle<Quote> quoteHandle[numberOfBonds];
    for (Size i = 0; i < numberOfBonds; i++) {
        quoteHandle[i].linkTo(quote[i]);
    }

    Frequency frequency = Annual;
    DayCounter dc = SimpleDayCounter();
    BusinessDayConvention accrualConvention = ModifiedFollowing;
    BusinessDayConvention convention = ModifiedFollowing;
    Real redemption = 100.0;

    Calendar calendar = NullCalendar();
    Date today = calendar.adjust(Date::todaysDate());
    Date origToday = today;
    Settings::instance().evaluationDate() = today;

    Natural bondSettlementDays = 0;
    Natural curveSettlementDays = 0;
    Date bondSettlementDate = calendar.advance(today, bondSettlementDays*Days);
    std::cout << "Today's date: " << today << std::endl;
    std::cout << "Bonds' settlement date: " << bondSettlementDate << std::endl;

    std::vector<boost::shared_ptr<BondHelper> > instrumentsA;//for FittedBondDiscountCurve
    std::vector<boost::shared_ptr<RateHelper> > instrumentsB;//for PiecewiseYieldCurve

    for (Size j = 0; j < LENGTH(lengths); j++) {
        Date maturity = bondSettlementDate;
        while (dc.yearFraction(bondSettlementDate, maturity) < lengths[j]) {
            maturity++;
        }

        Schedule schedule(bondSettlementDate, maturity, Period(frequency),
            calendar, accrualConvention, accrualConvention,
            DateGeneration::Backward, false);
    }
}

```

```

        boost::shared_ptr<BondHelper> helperA(
            new FixedRateBondHelper(quoteHandle[j],
            bondSettlementDays,
            FACE_VALUE,
            schedule,
            std::vector<Rate>(1, coupons[j]),
            dc,
            convention,
            redemption));

        boost::shared_ptr<RateHelper> helperB(
            new FixedRateBondHelper(quoteHandle[j],
            bondSettlementDays,
            FACE_VALUE,
            schedule,
            std::vector<Rate>(1, coupons[j]),
            dc,
            convention,
            redemption));
        instrumentsA.push_back(helperA);
        instrumentsB.push_back(helperB);
    }

    bool constrainAtZero = true;
    Real tolerance = 1.0e-10;
    Size max = 5000;

    ////////////////The main solution (in context of the task)////////////////////
    boost::shared_ptr<YieldTermStructure> ts0(
        new PiecewiseYieldCurve<ZeroYield, Linear>(curveSettlementDays,
        calendar,
        instrumentsB,
        dc));

    double d5_4 = ts0->discount(MATURITY);
    double d4_4 = ts0->discount(MATURITY - 1);
    double d3_4 = ts0->discount(MATURITY - 2);
    double d2_4 = ts0->discount(MATURITY - 3);
    double d1_4 = ts0->discount(MATURITY - 4);
    double d0_4 = ts0->discount(MATURITY - 5);
    double f = MATURITY - 5; //fraction of the 1st coupon
    double bond11PV
        = COUPON*(d5_4 + d4_4 + d3_4 + d2_4 + d1_4 + d0_4*f) + FACE_VALUE*d5_4;
    std::cout << "PV Bond11 (<ZeroYield, Linear> interpolation): " << bond11PV << "\n";

    ////////////////Alternative Solutions////////////////////

    //<Discount, LogLinear> is more common in practice than <ZeroYield, Linear>
    boost::shared_ptr<YieldTermStructure> ts1(
        new PiecewiseYieldCurve<Discount, LogLinear>(curveSettlementDays,
        calendar,
        instrumentsB,
        dc));
    d5_4 = ts1->discount(MATURITY);
    d4_4 = ts1->discount(MATURITY - 1);
    d3_4 = ts1->discount(MATURITY - 2);
    d2_4 = ts1->discount(MATURITY - 3);
    d1_4 = ts1->discount(MATURITY - 4);
    d0_4 = ts1->discount(MATURITY - 5);

```



```

f = MATURITY - 5;
bond11PV = COUPON*(d5_4 + d4_4 + d3_4 + d2_4 + d1_4 + d0_4*f) + FACE_VALUE*d5_4;
std::cout << "PV Bond11 (<Discount, LogLinear> interpolation): " << bond11PV << "\n";

//////////Svensson Fitting is used by German Central Bank (Bundesbank)//////////
boost::shared_ptr<FittedBondDiscountCurve> ts2(
    new FittedBondDiscountCurve(curveSettlementDays,
        calendar,
        instrumentsA,
        dc,
        SvenssonFitting(),
        tolerance,
        max));
d5_4 = ts2->discount(MATURITY);
d4_4 = ts2->discount(MATURITY - 1);
d3_4 = ts2->discount(MATURITY - 2);
d2_4 = ts2->discount(MATURITY - 3);
d1_4 = ts2->discount(MATURITY - 4);
d0_4 = ts2->discount(MATURITY - 5);
f = MATURITY - 5;
bond11PV = COUPON*(d5_4 + d4_4 + d3_4 + d2_4 + d1_4 + d0_4*f) + FACE_VALUE*d5_4;
std::cout << "PV Bond11 (Svensson interpolation): " << bond11PV << "\n";

//////////The same via bond pricing engine//////////
RelinkableHandle<YieldTermStructure> yieldTermStructure;
boost::shared_ptr<PricingEngine> bondEngine(
    new DiscountingBondEngine(yieldTermStructure));

Date maturity = bondSettlementDate;
while (dc.yearFraction(bondSettlementDate, maturity) < MATURITY) {
    maturity++;
}

Schedule schedule(bondSettlementDate, maturity, Period(frequency),
    calendar, accrualConvention, accrualConvention,
    DateGeneration::Backward, false);

FixedRateBond bond11(bondSettlementDays,
    FACE_VALUE,
    schedule,
    std::vector<Rate>(5, COUPON/100.0),
    dc,
    convention,
    redemption);

bond11.setPricingEngine(bondEngine);

yieldTermStructure.linkTo(ts0);
std::cout << "PV Bond11 (pricing engine, ts0): " << bond11.NPV() << "\n";

yieldTermStructure.linkTo(ts1);
std::cout << "PV Bond11 (pricing engine, ts1): " << bond11.NPV() << "\n";

yieldTermStructure.linkTo(ts2);
std::cout << "PV Bond11 (pricing engine, ts2): " << bond11.NPV() << "\n";
}

```

Code 3.2 Bootstrapping algorithms in a realistic case

But a surrounding smart pointer is allocated in stack, so its destructor is automatically called as the smart pointer gets out of scope. As soon as the destructor⁴² is called, the reference counter is decreased by one. And if it was the last reference, a delete on *px* is committed⁴³.

Further we see `RelinkableHandle<Quote> quoteHandle[numberOfBonds]`.

Handle-Pattern is important concept in QuantLib⁴⁴. Since it is much better to explain it by the example of term structure and pricing engine, we postpone it for a little bit.

Next new issue is `Settings::instance().evaluationDate() = today;`.

I believe, you have already got used to the fact that the advanced C++ features are intensively (ab)used in QuantLib: `evaluationDate()` returns a reference so the call of a this function on the left side of assignment expression is valid. In turn, `Settings::instance().` implement the Singleton pattern, which means that there is no more than one instance of settings. If need not necessarily dwell upon this technical details, it suffices just to learn how the evaluation date is set.

`Natural curveSettlementDays` are the number of curve settlement days. It is analogous to bond settlement days that we have already considered in Code 1.2.

Further we see

```
std::vector<boost::shared_ptr<BondHelper> > instrumentsA; //for FittedBondDiscountCurve
std::vector<boost::shared_ptr<RateHelper> > instrumentsB; //for PiecewiseYieldCurve
```

As you readily see, both vectors are actually filled with [shared pointers to] the objects of *FixedRateBondHelper* class. A *FixedRateBondHelper* just contains the bond [data] and its [clean] price. A vector of *FixedRateBondHelpers* is used to calibrate a yield curve. But why do we need to distinguish between *RateHelpers* and *BondHelpers* (actually, *BondHelper* even inherits from *RateHelper*)?! The reason is very simple: the fact that *BondHelper* inherits from *RateHelper* does not imply that `std::vector<boost::shared_ptr<BondHelper> >` inherits from `std::vector<boost::shared_ptr<RateHelper> >`. And we pass the vectors of [shared pointers to] helpers as an argument for the constructors of curve builders.

Finally, we set up the yield curve. We start with `PiecewiseYieldCurve<ZeroYield, Linear>` since linear interpolation is required in task (and we can guess that the linear interpretation of zero yields is much more plausible than that of discount factors). After the curve is calibrated we can calculate the discount factors and price our bond. Since we calculate the clean price, we consider only a fraction of the first coupon.

We repeat the same calculation with `PiecewiseYieldCurve<Discount, LogLinear>` (because it is probably the most popular piecewise-interpolated term structure in practice) as well as with `FittedBondDiscountCurve, SvenssonFitting` (because Svensson's approach is used by German Central Bank⁴⁵). As you see, the results are though similar but still different. And once again: there is no only right way to connect dots.

⁴² There is no destructor in `shared_ptr.hpp` but `boost::detail::shared_count` has a destructor, which is called in turn.

⁴³ If you want to trace this process in detail, have a look at `BOOST_SP_INTERLOCKED_DECREMENT` and `boost::checked_delete`

⁴⁴ See also <http://quantlib.org/slides/dima-ql-intro-1.pdf#Navigation81>

⁴⁵ We may have also used the cubic splines, like central US and Japanese banks do.

And additionally, we reproduce our calculation with `DiscountingBondEngine`. In QuantLib it is generally so that in order to price an instrument we need to attach a pricing engine to this instrument (we do it by means of `bond11.setPricingEngine(bondEngine)`). In turn, virtually every pricing engine needs a yield term structure. Or, better to say, a Handle to a term structure. Now we have a good opportunity to explain the Handle-Pattern.

In real world the yield change rapidly. And there may be hundreds of very different instruments and thus hundreds of pricing engines that, however, depend on the same term structure. Thus it would be too tedious to set a new term structure by each pricing engine.

Alternatively, we could make a term structure as a tuple of quotes and use Observer/Observable pattern. Of a quote changes, it notifies its Observer, i.e. a term structure, which in turn notifies its Observers, i.e. the pricing engines. Well, updating a dozen of quotes seems to be a better idea than updating of hundreds of engines but there is an even better idea. We introduce Handle, a mediator between the pricing engines and the term structure. A Handle notifies its pricing engine about updates of the term structure. In turn, this update mechanism (a.k.a. relinking) is implemented in RelinkableHandle, which is derived from Handle. Thus we create an RelinkableHandle, upcast it to Handle and set this Handle to the pricing engines. The engines will be notified of the term structure updates but they cannot relink the Handle to a new term structure. We, however, can (as soon as we keep an reference to the original RelinkableHandler). Actually, this is what we do in the last section of Code 3.2.

Chapter 4: Stochastic interest rate models

In the previous chapter we discussed how to fit a yield curve to a set of bonds. This step, i.e. fitting the (current) term structure, is necessary for nearly every financial calculation. But this is, so to say, a static view. In turn, the stochastic interest rate models let us model the *dynamics* of the yield curve. For a (simple) risk management, we need to model the evolution of the interest rates under the real-world measure: e.g. if we want to calculate the VaR of a portfolio of (straight) bonds. But in order to price an interest rate derivative we need a model, calibrated under the martingale measure. Finally, there are the use cases in which we need the both measures: for instance, if a portfolio contains the interest rate derivatives and we want to calculate the VaR, we first simulate portfolio dynamics under the real-world measure (scenario generation) and then we switch to the risk-neutral measure in order to calculate the portfolio value for each scenario.

The most common in practice are the short rate models and the LIBOR market model. If you are looking for mathematical details you should read my tutorial, which is freely available in Internet⁴⁶. Here I will just provide a quick summary. As is well known, in practice the shortest borrowing is the overnight borrowing (intraday borrowing are also possible but they are generally interest-free). It is also known that the central banks influence the short term rate but actually target the whole term structure. So the short rate models try to explain the evolution of the yield curve by the dynamics of the instantaneous spot rate (a.k.a. short rate).

But you should be careful: short rate is not an overnight rate! Rather it is an *unobservable* quantity, a mathematical abstraction. In particular, Damir Filipovic warns against a (straightforward) usage of the overnight rate time series for the calibration of a short rate model “because the motives and needs driving overnight borrowers are very different from those of borrowers who want money for a month or more”⁴⁷. Model calibration is in general very challenging issue but in case of the short rate models it is particularly intricate and often insufficiently elucidated in MFE courses.

Usually one gets started with the Vasicek model

$$dr_t = (a - br_t)dt + \sigma dW_t$$

In original paper⁴⁸ Vasicek worked only with the real-world probability measures (in 1977 the risk-neutral pricing was not yet discovered). He constructed a risk-free portfolio (just like Black and Scholes did⁴⁹) and used no arbitrage principle to derive the formula for a zero bond price. Later it was shown that the short rate dynamics under the martingale measure may be described with the same equation (with different drift but the same volatility). One can switch from the drift und the real-world measure to that under the martingale measure by means of the market price of risk λ .

So if there is a closed-form solution for the price of a zero bond can we calibrate the Vasicek model to the (current) yield curve? In principle, yes - we just need to choose the model parameters that minimize the distance⁵⁰ between the theoretical and market bond prices. Moreover, this will be a calibration directly under the risk neutral measure⁵¹. Note that in order to calibrate the stochastic *process* that drives the short

⁴⁶ <http://www.yetanotherquant.com/libor/tutorial.pdf>

⁴⁷ Damir Filipovic - Term Structure Models: A Graduate Course - Springer, 2009 (p. 10)

⁴⁸ Notably, this paper is likely more often cited than read
(<http://www.wilmott.com/messageview.cfm?catid=4&threadid=85948>)

⁴⁹ Actually, it was Merton, who did.

⁵⁰ E.g. the sum of squares.

⁵¹ If you want to calibrate the model under the real-world measure you can either ignore Filipovic's warning and use a time series of the overnight rates as a proxy for the short rate or use more advanced approach like Kalman filter (see

rate one does not need a time series, rather a "static snapshot" (i.e. the current term structure) is enough. At the first glance it seems surprising but it is plausible: a bond price $P(t, T)$ represents the expected evolution (und the martingale measure) of the short rate over the *whole* period (t, T) .

However, the Vasicek model has a big problem, it is *not term structure consistent*, i.e. we cannot fit the model parameters in such a way that the empirical and theoretical bond prices coincide. Most likely this is the reason why Vasicek model is rarely used in practice. In particular, in QuantLib it seems to be just a stub for the Hull White model, which is term structure consistent. But before we discuss this model in detail it is worth getting a holistic view on models, implemented in QuantLib. Have a look at Figure 4.1. On the left from the dotted line there are "classical" models that are considered in the textbooks on term structure modeling⁵². The term structure consistent models are marked with T.

First, all (concrete) classes are derived from `QuantLib::CalibratedModel` which is obvious, since a non-calibrated model is useless in practice. But interestingly, the calibration logic is encapsulated in the basis class (except `QuantLib::MarkovFunctional`, where the method `calibrate()` is overridden).

Second, all short rate models except G2 are one factor. It is actually a pity because it is commonly agreed that one needs three factors in order to realistically model a yield curve: level, slope and curvature.

e.g. <http://www.bankofcanada.ca/wp-content/uploads/2010/02/wp01-15a.pdf>). Note that λ is not (directly) observable.

⁵² As to the right-hand side, have a look at <http://quantlib.org/slides/qlws13/caspers.pdf>

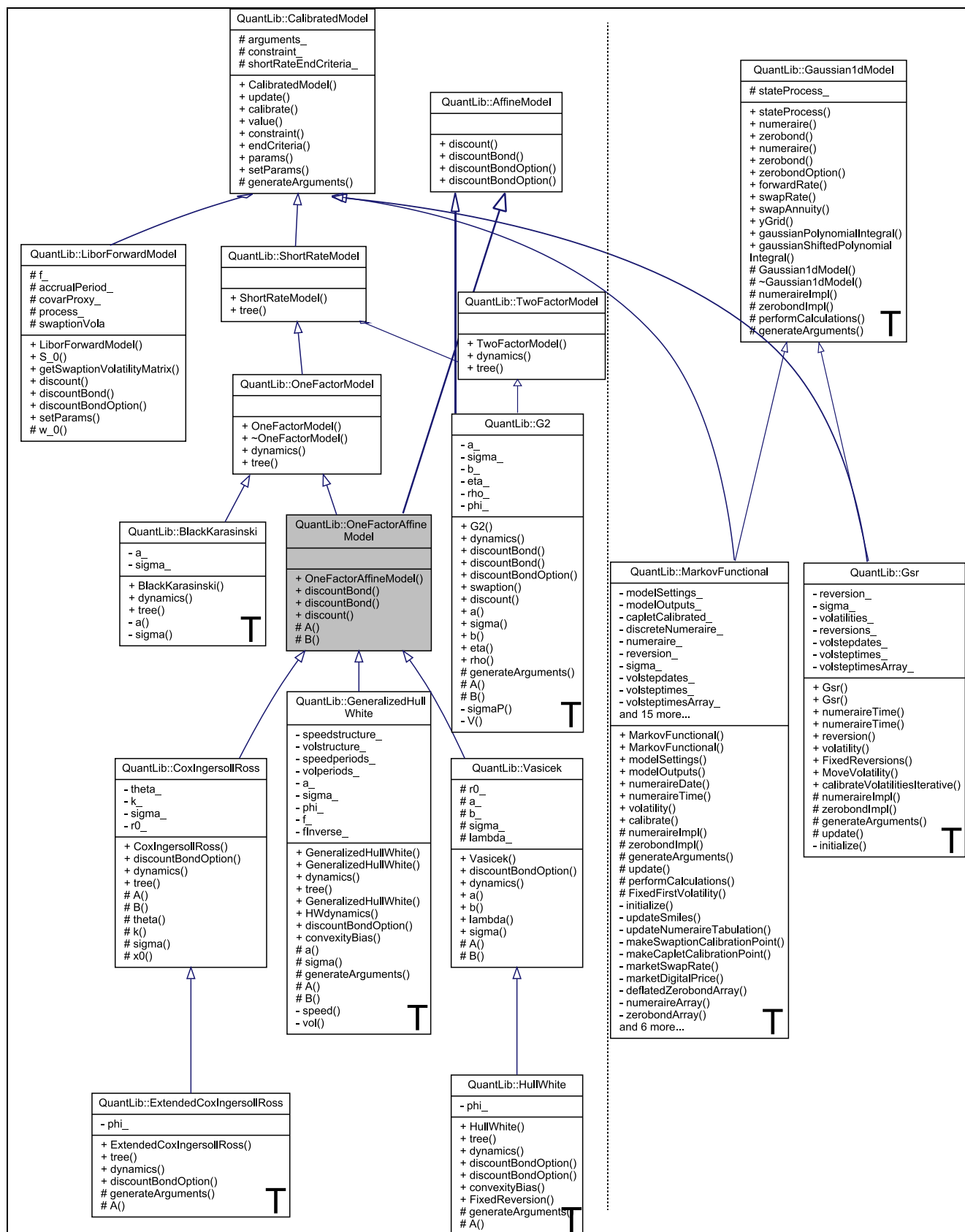


Figure 4.1 A (more or less) complete overview of the stochastic interest rate models in QuantLib 1.4.1

Appendix A. Refactoring QuantLib::Date

Since it is very uncomfortable for debugging to work with `serialNumber_` representation of *Date*, we try to refactor this class. Since it would be very cumbersome to redesign it from scratch, our idea is as follows: we rename the current implementation of *Date* to *DateImpl* and then define a new class *Date* that will have all public methods the previous version of *Date* had and three member variables *Day day_*, *Month month_* and *Year year_*. In all public methods we will create temporal instance(s) of *DateImpl*, call the respective methods of *DateImpl* and, if necessary, update the member variables of *Date*.

For example, the implementation of the method `Date::month()` will be as follows

```
Month Date::month() const
{
    DateImpl d(day_, month_, year_);
    return d.month();
}
```

Finally, we will write a respective `natvis-Visualizer`.

At the first glance it is a perfect idea: with little efforts we expect to get the identical functionality (probably, a little bit slow, but it does not matter). All we expect to do is to re-implement the public methods in a pretty trivial way, leaving the implementation of the previous *Date* (which is now *DateImpl*) intact. However, the practice is more complicated. First of all the files *date.hpp* and *date.cpp* contain not only the declaration and the definition of *Date* but also some auxiliary structures like *struct short_date_holder*, *struct long_date_holder*, etc (look for namespaces *io* and *detail*). They are used to print the objects of the *Date* class to console.

At the first glance there is no problem in context of our approach. However, the old version of *Date* (which is now *DateImpl*) uses the functionality of these auxiliary structures in a private method *DateImpl::checkSerialNumber(BigInteger serialNumber)*.

```
void DateImpl::checkSerialNumber(BigInteger serialNumber) {
    QL_REQUIRE(serialNumber >= minimumSerialNumber() &&
        serialNumber <= maximumSerialNumber(),
        "Date's serial number (" << serialNumber << ") outside "
        "allowed range [" << minimumSerialNumber() <<
        "-" << maximumSerialNumber() << "], i.e. [" <<
        Date::minDate() << "-" << Date::maxDate() << "]" );
}
```

In the last line we would, according to our idea, write `DateImpl::minDate() << "-" << DateImpl::maxDate()` but we cannot (or we need **additionally** define the output of *DateImpl* to the console)...

Well, using only *Date::minDate()* and *Date::maxDate()* in a private method of *DateImpl* is actually a small sin, which we can commit without remorse.

However, there is another problem: the range of valid dates is between 01.01.1901 and 31.12.2199 but there is a special case: the null date. It is created when one calls the default constructor *Date()*, which is now *DateImpl()*. One can apply the operators "=", "<", "<=", ">", ">=" to the null dates but cannot call the methods *dayOfYear()*, *month()*, *year()* without causing an exception.

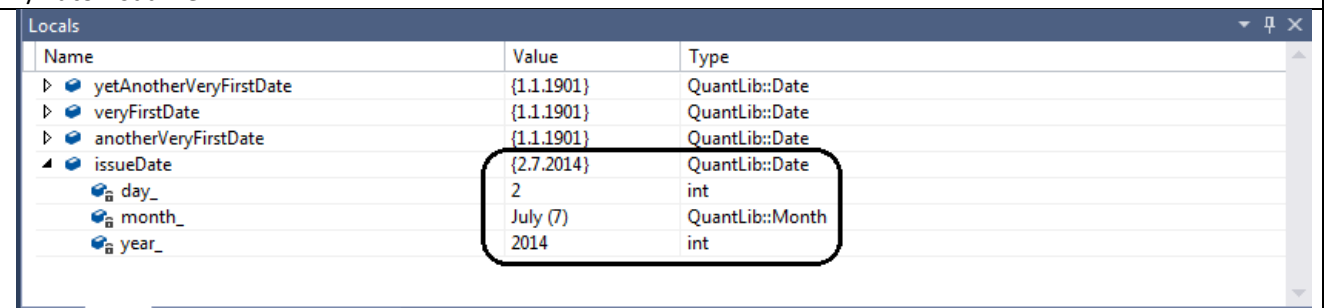
Thus a nice attempt to re-implement e.g. the operator ">=" like this

```
inline bool operator>(const Date& d1, const Date& d2) {
    DateImpl di1(d1.dayOfMonth(), d1.month(), d1.year());
    DateImpl di2(d2.dayOfMonth(), d2.month(), d2.year());
    return (di1.serialNumber() > di2.serialNumber());
}
```

will fail because the attempt to call *dayOfYear()*, *month()*, *year()* in case of a null date throws an exception... A rude practical solution is to re-define the idea of the null date (actually, the default date). I decided to set the null date equal to the mindate, i.e. to 01.01.1901. This allows a straightforward re-implementation of all methods without a cumbersome consideration of the null date(s) case.

You need to put the file *QuantLib_Date.natvis* to " \Documents\Visual Studio 2013\Visualizers"

```
<?xml version="1.0" encoding="utf-8"?>
<AutoVisualizer xmlns="http://schemas.microsoft.com/vstudio/debugger/natvis/2010">
<Type Name="QuantLib::Date">
  <DisplayString>{{{day_}.{{(int)month_}.{year_}}}</DisplayString>
</Type>
</AutoVisualizer>
```



Name	Value	Type
▶ yetAnotherVeryFirstDate	{1.1.1901}	QuantLib::Date
▶ veryFirstDate	{1.1.1901}	QuantLib::Date
▶ anotherVeryFirstDate	{1.1.1901}	QuantLib::Date
▲ issueDate	{2.7.2014}	QuantLib::Date
day_	2	int
month_	July (7)	QuantLib::Month
year_	2014	int

Natvis specification for the refactored QuantLib::Date

Finally, don't forget to run the QuantLib testsuite⁵³. The refactoring case we have considered is an excellent justification of the unit test usefulness. The testsuite is pretty extensive, so if all tests run successfully we may be quite sure that our refactoring was correct. But since we changed the definition of the null date, a couple of tests do fail. Still these are very special cases, so we will further use the refactored version of *QuantLib::Date* in this book. But it is not recommended to use it in productive software!

Wanna get the source code of the modified
QuantLib::Date ?!

[Donate \\$10 to the author ;\)](#)

⁵³ NB! I may take a couple of days to complete!