Software Frameworks in Quantitative Finance

Part II: Developing C++ Applications for the Finite Difference Method (FDM)

Daniel J. Duffy

Abstract
In this article we discuss a number of techniques that help us design and implement flexible and maintainable software systems for option pricing models. We concentrate on showing how to model the Finite Difference Method (FDM) using C++ in combination with design patterns and software frameworks. In particular, we describe how the framework can be used as a design tool that allows us to create new pricing models and schemes by customizing the framework in a non-intrusive way.

We assume that the reader has some knowledge of design patterns. We have discussed these in previous editions of Wilmott; in particular, the current article is a generalization of the results in Duffy 2005.

1. Background, Overview and Goals

In this article we discuss how to design and implement a software application for a class of option pricing models. For convenience, we concentrate on a one-factor plain option problem; the formulation subsumes a range of option types such as barrier, lookback and standard European options. This approach allows us to focus on the essential problems. Once we have designed and implemented the code for the one-factor problem we shall see that the generalisation to more complex cases becomes feasible.

To be precise, let us consider a one-factor plain option model that we describe by a second-order parabolic partial differential equation defined on a bounded interval (A, B) in space and on a bounded interval (0, T) in time. The partial differential equation is augmented by boundary conditions (for convenience we take Dirichlet boundary conditions) and initial condition (this corresponds to the payoff function in finance). In order to accommodate a range of option models we describe the so-called initial boundary value problem as follows:

$$Lu \equiv -\frac{\partial u}{\partial t} + \sigma(x,t)\frac{\partial^2 u}{\partial x^2} + \mu(x,t)\frac{\partial u}{\partial x} + b(x,t)u = f(x,t) \text{ in } D$$

In this case we need to find a function u = u(x,t) that satisfies the partial differential equation as well as the boundary and initial conditions. Please note that the variable x is a general parameter and does not (necessarily) denote the logarithm of the asset price as is usual in finance and furthermore, the time variable t is defined in the 'forward' direction (that is, from t = 0 to t = T). It is a question of notation, but it can lead to some confusion. We need to be aware of these facts when examining system (1) and what the symbols and notation in it really mean. In short, system (1) is a very generic initial boundary value problem and it can be *instantiated* to produce many examples of specific initial boundary value problems in quantitative finance.

It is not difficult to write code that implements numerical methods to approximate the solution of initial boundary value problems such as system (1), for example. However, many ad-hoc implementations lead to sub-optimal solutions and for this reason we wish to define in how far the software can be customized to suit new requirements. To this end, we draw up a list of features that the new design should satisfy:

- Requirement 1: It must be possible to model a wide range of initial boundary value problems of type (1) above. In particular, we model the Black Scholes equation using a variety of payoff functions and option types. Furthermore, the software must support constant volatility and time-dependent volatility surfaces. Finally, it must be possible to enter parameters (such as interest rates and maturities) from dialog boxes, databases and real-time data feeds.
- Requirement 2: We wish to implement new finite difference schemes for system (1) quickly and with the least amount of effort. The rationale for this requirement is that we would like to apply many finite difference schemes to solve initial boundary value problems and it is in our interest to do this without having to make major changes in the software.

- Requirement 3: It should be possible to visualize the output from the finite difference engine. It particular, we are interested in presenting option prices at the maturity date (and possibly other dates) in Excel. We may even wish to adapt the software so that it can be used as a COM addin. In this latter case the user dialog takes place in Excel itself.

We now describe how to realise these goals using modern software design techniques.

2. Modelling Options using PDE and FDM

In this section we show how to model the initial boundary value problem (1) and the related finite difference schemes that approximate it. Furthermore, we ensure that the three requirements of section one are realised in our design.
In general, each well-defined unit of functionality will be implemented as a class or some kind of class hierarchy.

2.1 Partial Differential Equations for Option Models

We are able to model a wide range of option models because we have designed the system (1) using the *Bridge* pattern. We recall that this pattern associates a generic class with a specific implementation thereof. In the current case the generic class implements the components in system (1) while we can choose how to implement this class using other application-specific classes. We depict the design as an UML class diagram in figure 1.
The class `IBVP` contains the interface specifications that implement system (1) while `Implementation` is the base class for all specific implementations, such as class `BlackScholes`, for example. In the latter case we see this class uses the services of a class that models one-factor options in conjunction with payoff functions. This is a very flexible design because:

- Clients of `IBVP` have no knowledge of its low-level implementation (this is called *information hiding*)
- We can add new classes to the class network simply by implementing a number of pure virtual functions
- Delegation and Composition principle: delegation is sometimes called *object-level inheritance*; in the current context this means that we can switch implementations at run-time. This is valid for both the *Bridge* and *Strategy* patterns in figure 1.

We now give an indication of how we implemented the C++ code that realizes the UML diagram in figure 1. First, the interface specification for the class `IBVP` is a direct mapping of the initial boundary problem in system (1) to code and is given by:

```
class IBVP
{
private:
  Range<double> xaxis; // Space interval
  Range<double> taxis; // Time interval

  IBVPImp* imp;        // Bridge implementation
```

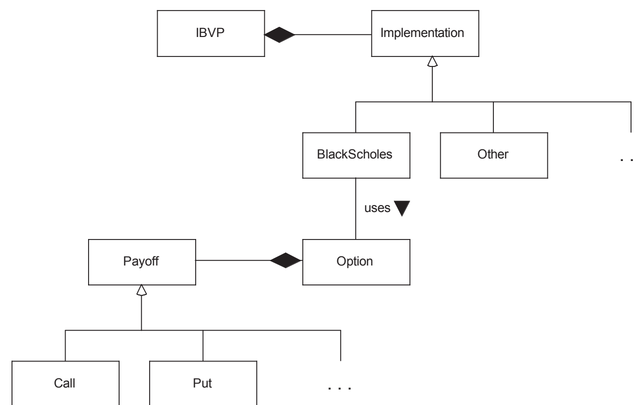

Figure 1 PDE Class Network

```
private:
  IBVP();
public:
IBVP (IBVPImp& executor, const Range<double>& xrange,
               const Range<double>& trange);

// Selector functions
// Coefficients of parabolic second order operator
double diffusion(double x, double t) const; // Second derivative

double convection(double x, double t) const;

double zeroterm(double x, double t) const;

double RHS(double x, double t) const;

// Boundary and initial conditions
double BCL(double t) const;   // Left hand boundary
                              // condition
double BCR(double t) const;   // Right hand boundary
                              // condition

double IC(double x) const;    // Initial condition


// The domain in which the PDE is 'played'
Range<double>& xrange();
Range<double>& trange();

};
```

These functions will be implemented by specific classes such as a class that models the one-factor Black Scholes PDE; this gets it data from a class `Option` which in its turn has a corresponding `Payoff` pointer (this is an instance of a *Strategy* pattern):

```
class BlackScholes : public Implementation
{
public:

  Option* opt;

  BlackScholes(Option& option);

  double diffusion(double x, double t) const;


  double convection(double x, double t) const;
```


© Datasim Financial, 2007

```
      // etc.


  double IC(double x) const;

};
```

The class for the European option is a data container and it has an embedded payoff object. The (simple) base class for all kinds of derivatives is given by:

```
class Instrument
{
public:

};
```

and the current option class' code is given by:

```
class Option : public Instrument
{
public:
  // PUBLIC, PURE FOR PEDAGOGICAL REASONS
  double r;    // Interest rate
  double sig;  // Volatility
  double K;    // Strike price
  double T;    // Expiry date
  double b;    // Cost of carry
  double SMax; // Far field condition
  char type;   // Call or put

// An option uses a polymorphic payoff object
  Payoff* OptionPayoff;

  Option()
  {

  }
};
```

One final remark concerning the structure of the classes that implement the Black Scholes model in figure 1; we note that the configuration is an example of a *Layers* pattern (Buschmann 1996, Duffy 2004). In this case we have designed the problem by partitioning the current implementation into three layers:

- Layer 1 (Control layer, in this case `BlackScholes`): the layer that implements the services required by its clients (for example, the `FDM` class as described in the next section)
- Layer 2 (Data or Entity layer): contains application objects, in this case the class `Option`
- Layer 3 (Boundary or hardware layer): the layer that generates the data for the Entity layer. We shall discuss this issue in more detail in section three where we introduce various factory patterns

This discussion completes the description of the so-called continuous problem. The full source code can be found in Duffy 2006a.

2.2 Finite Difference Schemes

We now discuss how to approximate system (1) using finite difference schemes. We concentrate on the main design issues here because the underlying theory and practical coding issues have been discussed elsewhere (see Duffy 2006, Duffy 2006a and Duffy 2005). Our main interest lies in being able to test and install new finite difference schemes with as little code change as possible. We achieve this by defining standard interfaces between classes (using *virtual* functions, for example) and decomposing this part of the class network into independent components, as can be seen in figure 2.

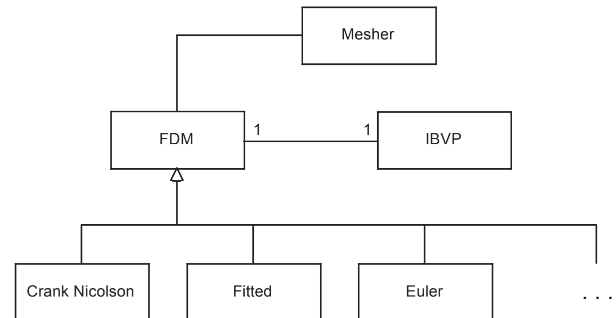

Figure 2 FDM Class Network

An extremely important design pattern that promotes framework construction is the *Template Method Pattern* (Gamma 1995). This pattern is concerned with algorithm design and implementation; the algorithm consists of an *invariant part* that is common to all (derived) classes and a *variant part* that must be defined in derived classes. In the current case the invariant part could correspond to mesh generation (among other things) while the variant part corresponds to how specific finite difference schemes (for example, the Euler or Crank Nicolson schemes) actually compute the solution at time level n+1 in terms of the known data at time level n. To this end, the interface for the base class `FDM` is given by:

```
class FDM
{  // Set of finite difference to solve
   // scalar initial boundary value problems

private:

   // …


protected:

   IBVP* ibvp;  // Pointer to 'parent'

   long N;   // The number of subdivisions of inter-
val
   double k; // Step length in time
   long J;   // The number of subdivisions of inter-
val

   // …

   Mesher m;
   Vector<double, long> xarr; // Useful work array
   Vector<double, long> tarr; // Useful work array

   // …
```

```
  // Other data
  long n;      // Current counter
  Vector<double, long> vecOld;
  Vector<double, long> vecNew;

  FDM(const FDM& source);
  FDM& operator = (const FDM& source);

public:
  FDM();
  FDM(IBVP& source, long NSteps, long JSteps);
  virtual ~FDM();

  The result of the calculation
  NumericMatrix<double, long>& result();

  // Mesh data
  Vector<double, long> XValues() const;
                              // Array of x values
  Vector<double, long> TValues() const;
                              // Array of time val-
ues

  // Hook function for Template Method pattern
  virtual void calculateBC() = 0;
                              // Calculate BC at n+1
  virtual void calculate() = 0;
                              // Calculate sol. at
n+1

};
```

Derived classes must implement these last two functions and then we are done in the sense that we can test our new finite difference schemes almost immediately. The full code for the algorithm is given by:

```
NumericMatrix<double, long>& FDM::result()
{ // The result of the calculation

L1:
  tnow = tprev + k;


  // Template Method pattern here

  // Variant part
  calculateBC(); // Calculate the BC at n+1

  // Variant part; pure virtual hook functions
  calculate (); // Calculate the solution at n+1

  // Invariant part
  if (currentIndex < maxIndex)
  {

       tprev = tnow;
       currentIndex++;
       // Now postprocess
       res.Row(currentIndex, vecNew);
       vecOld = vecNew;

       goto L1;
  }

  return res;
}
```

We describe this code in words; this code is a simple state machine in which the option price at level n+1 is calculated. The main steps are: increment the time level, calculate the boundary conditions, calculate the solution at level n+1 (each kind of scheme has its own way of doing this); finally, the invariant part of the code corresponds to adding the solution at time level n+1 to the matrix denoted by the variable `res`.

3. Initialisation, Configuration

We have discussed the class diagrams and structural relationships between classes in the previous sections. In order to 'complete the jigsaw' we need to discuss how the application is configured and how the data in the application is initialised. In short, we discuss:

- *Abstract Factory* pattern that initialises the option data
- The *Builder* pattern that configures all the classes in figures 1, 2 and 3

The first pattern is shown in figure 3. Here we have a number of factory classes for creating option data, for example using the console or by some remote data feed system from different vendors or providers. We focus on showing how to generate the data using the console. To this end, the abstract interface that creates an option is realized by an abstract class:

```
class InstrumentFactory
{
public:

  virtual Option* CreateOption() const = 0;

};
```

The class for console input is defined as follows:

```
class ConsoleInstrumentFactory : public Instrument-
Factory
{
public:

  Option* CreateOption() const
  {
```

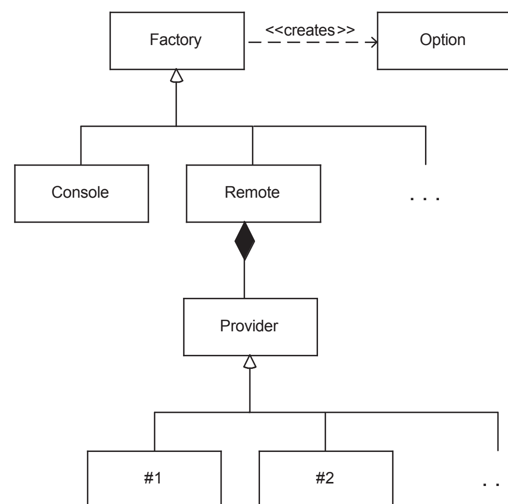

Figure 3 Creational Patterns

© Datasim Financial, 2007

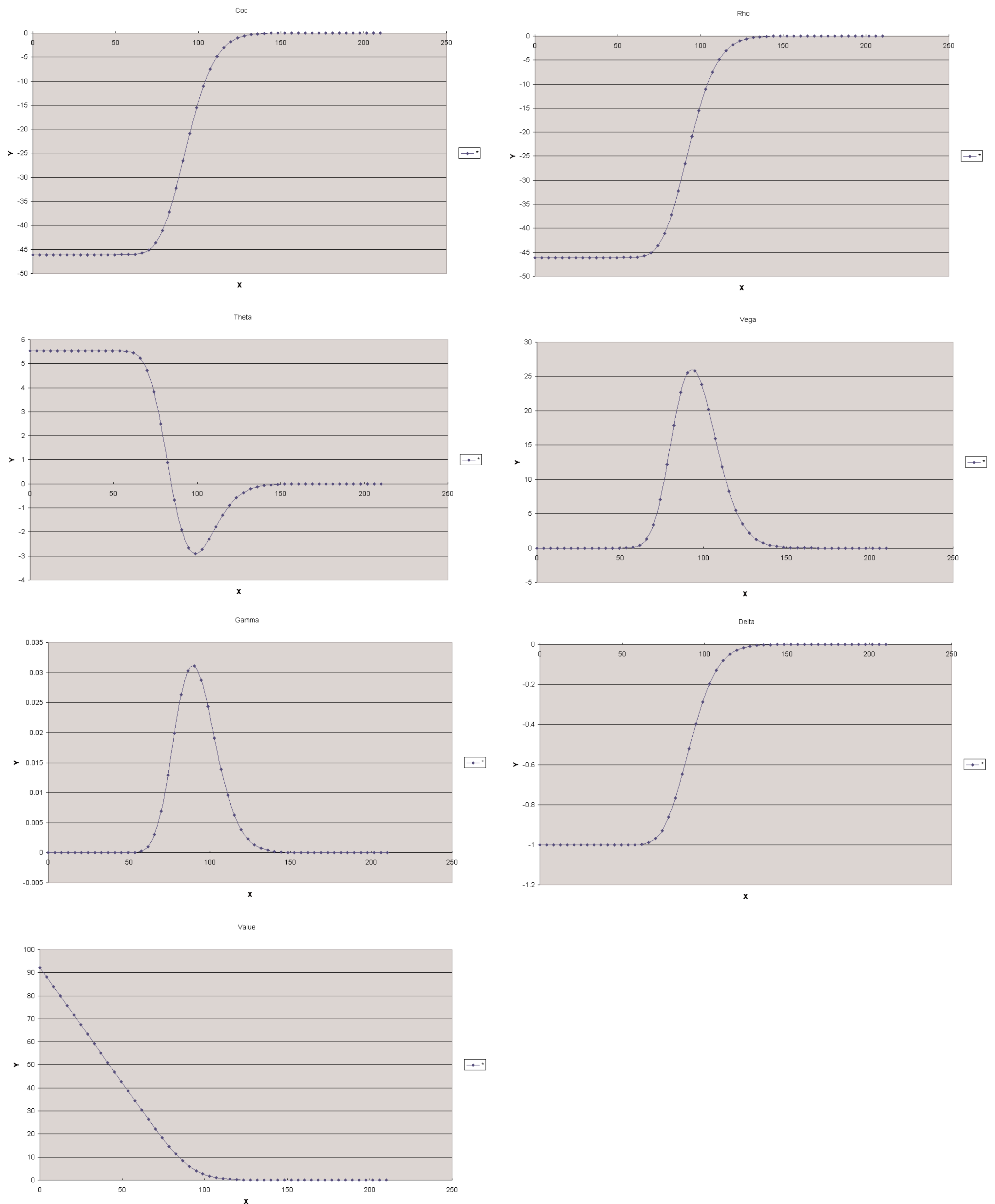

Figure 4 Visualisation in Excel

```
        double dr;       // Interest rate
        double dsig;     // Volatility
        double dK;       // Strike price
        double dT;       // Expiry date
        double db;       // Cost of carry
        double dSMax;    // Far field boundary

        cout << "Interest rate: ";
        cin >> dr;

        // more input here for the other parameters

        Option* result = new Option;

        cout << "Payoff 1) Call, 2) Put: ";
        int ans;
        cin >> ans;
        if (ans == 1)
        {
              result-> type = 'C';
              (*result).OptionPayoff =
                 OneFactorPayoff(dK, MyCallPayoffFN);
        }
        else
        {
              result-> type = 'P';
              (*result).OptionPayoff =
                 OneFactorPayoff(dK, MyPutPayoffFN);
        }

        result->r   = dr;
        result->sig = dsig;
        result->K   = dK;
                result->T      = dT;
                result->b      = db;
                result->SMax = dSMax;

        return result;

  }
};
```

Finally, we employ the *Builder* pattern to construct the complete class network that is the union of the diagrams in figures 1, 2 and 3. In general, this pattern builds a complex object in an incremental manner while hiding details of the construction process from clients (see Gamma 1995). We have generalized it to support arbitrary class networks. It is possible to configure an application without having to resort to builders but the code tends to become unmaintainenable. Again, this pattern then becomes responsible for one main task, namely creation and initialization of classes and the relationships between them.

4. Examples and Applications

We now describe how to use the framework. To this end, we describe the interface functions that we can use in a main program and how the generated data are displayed in Excel, for example. We use the Excel visualization package that allows us to display output (for example, vectors and matrices) in a variety of formats. In particular, we can display the full set of option values for each stock price from time t = 0 to t = T. The code that realizes this output is given by:

```
cout << "Output: 1) Cell(matrix), 2) list of line
graphs: ";
int choice; cin >> choice;

// N.B. 'fdm' is an instance of class CrankNicolson,
for example

if (choice == 1)
{
  try
  {
       printMatrixInExcel(fdm.result(),
             fdm.TValues(),
             fdm.XValues());
       }
       catch (DatasimException& e)
       {
             e.print();
             return 0;
       }
  }

if (choice == 2)
{
  cout << "Frequency of output: "; int f; cin >> f;

  try
  {
       printMatrixChartsInExcel(f,
             fdm.result(),
       fdm.TValues(),
             fdm.XValues());

                 }
       catch (DatasimException& e)
       {
             e.print();
             return 0;
       }
  }
}
```

The results are then displayed as an Excel chart or as cells. To show what is possible we take a somewhat more extended diagram in which we display option price and its sensitivities for a range of values of the underlying S. The author finds this extremely useful when testing and debugging new schemes.

In this example we start Excel application (using COM) from our C++ code and we create a separate sheet for each kind of information that we are interested in viewing.

5. Extending the Framework

In this article we introduced a framework for one-factor option pricing problems. The architecture is reasonably generic and it can be applied to a range of problems. However, it can be improved in a number of ways, as already discussed in Kienitz 2006. First, we wish to decouple the classes `FDM` and `IBVP` even further so that these classes will have no knowledge of each other. We realize this requirement by introducing an object between these two classes. This will be an instance of a *Mediator* pattern. The main advantage of this design tactic is that we are forced to think in terms of standard interfaces and components. Another pattern is the *Blackboard* (as discussed in Kienitz 2007).

© Datasim Financial, 2007

5.1. More general one-Factor Pricing Models

There are a number of new requirements that will need to be realized if the system is to be useful for a wider range of one-factor models. Some important ones are:

- R1: Early exercise features and American options
- R2: Modelling jumps (for example, the Merton model)
- R3: Systems of equations (for example, convertible bonds)
- R4: Discrete and continuous monitoring
- R5: Integration with calibration software
- R6: Nonlinear problems and problems with discontinuities

Each of these requirements can be mathematically and numerically modeled using PDE and finite difference theory and then mapped to knowledge sources that will be integrated into the designs as shown in figures 1, 2 and 3. To take an example, to model R2 we need to refactor the class `IBVP` so that it can support not only partial differential equations but also partial-integro differential equations (PIDE). One way to do this is to use the *Whole-Part* pattern (Buschmann 1996) to aggregate the existing classes for the PDE part of the problem with the new classes for the PIDE part.

5.2. Does the Framework generalize to n Factors?

An important question is whether the framework can be modified, copied or extended in order to price two-factor and three-factor pricing problems, for example. The answer is yes. However, we propose using a generic approach by using C++ template mechanism. In general we set up classes in the framework in such a way that the dimension N of the problem is explicitly modeled as a template parameter. Then, when creating code for the two-factor problem all we have to do is to instantiate the template class by setting N to the value 2.

5.3. Performance and Multi-threading Applications

Many production applications have been designed and implemented to run on sequential machines but at the moment of writing we are witnessing the emergence of new hardware that can dramatically improve the performance of these applications. There are a number of issues to be addressed, such as:

- How do we design multi-threaded and high-performance applications?
- How do I migrate my current application to a multi-threaded one?

To partially answer the first question we note that there must a good match between the 'paper design' of a software system and its eventual implementation in hardware. For example, by using the *Mediator* pattern we create a central component that acts as a kind of hub between other loosely-coupled components. This design is very flexible and it also allows us to map it to software libraries that implement a *master-slave threading model*; the master is the main thread and it is responsible for the creation of other threads, one thread for each component. A library that supports that mode of operation is OpenMP (see www.openmp.org) and is a collection of compiler directives, library functions and environmental variables that can be used to specify shared-memory parallelism in C and C++ programs. One of the advantages of this library is that its directives can be included in existing code in a non-intrusive manner. This tactic constitutes one approach to improving the performance of existing applications but in general a partial redesign of critical parts of the application will be necessary.

Summary and Conclusions

We have given an overview of a customizable software framework in C++ that we have constructed by piecing design patterns into a network of cooperating objects. We emphasized the importance of partitioning the application into a hierarchy of loosely coupled software modules or components.

The advantages of this approach are that the software can be extended to suit new requirements and that it is suitable for a migration to a hardware environment containing multiple processors.

References

Alexandrescu, A. 2001 *Modern C++ Design: Generic Programming and Design Patterns Applied* Addison-Wesley

Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad and M. Stal 1996 *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley and Sons, Chichester, UK

Duffy, Daniel J. 2004 *Domain Architectures: Models and Architectures for UML Applications*, John Wiley and Sons, Chichester, UK

Duffy, Daniel J. 2004a *Financial Instrument Pricing in C++*, John Wiley and Sons, Chichester, UK

Duffy, Daniel J. 2005 *Design Patterns in Option Pricing Part III; Modelling the Finite Difference Method in C++* Wilmott Magazine July 2005

Duffy, Daniel J. 2006 *Finance Difference Methods in Financial Engineering, A Partial Differential Equation Approach*, John Wiley and Sons, Chichester, UK

Duffy, Daniel J. 2006a *Introduction to C++ for financial engineers, An Object-Oriented Approach*, John Wiley and Sons, Chichester, UK

Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1995 *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley Reading MA

Kienitz, J. and Duffy, Daniel J. 2007 *Software Frameworks in Quantitative Finance, Part I Fundamental Principles and Applications to Monte Carlo Methods* Wilmott January

Shaw, M. and D. Garlan 1996 *Software Architectures Perspectives on an emerging Discipline* Prentice Hall

Stroustrup, B. 1997 *The C++ Programming Language* (3rd Edition), Addison-Wesley Reading MA

© Datasim Financial, 2007

© Datasim Financial, 2007