Application Design in C# The Monte Carlo Method for Option Pricing

1. Introduction and Objectives

In this chapter we analyse, design and implement a software system to price one-factor options using the Monte Carlo method. The programming language used is C# and many of the design principles are also applicable to C++ and Java. For more background information on the Monte Carlo method and its applications to computational finance, see Glasserman 2004, Duffy and Kienitz 2009. The focus is on software design and implementation and we also discuss some of the mathematical and financial foundations relating to the Monte Carlo method. An understanding of this material is necessary if we are to understand the design rationale and the component interfaces.

The prerequisites for understanding this chapter are:

- Some acquaintance with stochastic differential equations (SDEs) and their numerical approximation by one-step finite difference schemes. We model these mathematical concepts as classes with well-defined responsibilities and interfaces. For background information, see Kloeden and Platen 1995.
- Some basic knowledge of plain, barrier, lookback and Asian options. The pricing modules model some of their essential properties.
- Knowledge of the software structure of a Monte Carlo engine (Duffy and Kienitz 2009). This structure will form the core of a (mediator) class that coordinates the data and control flow between the various modules in the application.
- System/functional decomposition techniques that produce a system context diagram as well as a set of loosely coupled and cohesive components. Additionally, knowing that the software system in this chapter is a special case of a *Domain Architecture* (Duffy 2004), namely an instance of a RAT (*Resource Allocation and Tracking*) system will give us insights into the structure of the current application (*reasoning by analogy*).
- Some high-priority design patterns, in particular the *Builder* pattern that we discussed in previous chapters; in general, we do not need the traditional GOF patterns in this application because of the defined process and system decomposition techniques that we employ. In fact, our design is more flexible in our opinion than any design based on the bottom-up object-oriented programming model in combination with the GOF patterns.
- Knowledge of some advanced C# features and the .NET Framework, in particular: delegates, tuples, generics, interfaces and the *Standard Event Pattern*.
- Knowledge of *policy-based design* and *provides-requires* interfaces. In particular, it is important to understand the concept of a *service* that a software module expects or requires from other modules. These are concepts that are not present in the traditional object-oriented model.

We have a number of goals that we would like to realise in this chapter. First, we introduce a number of techniques that help us design stable and flexible software systems. Second, we design an application from beginning to end to show how these design principles work. Finally, the lessons learned can be applied to other kinds of problems.

2. Problem Description

We have addressed and solved this problem using the C++ object-oriented model in Duffy and Kienitz 2009 as well as a new systems approach as shown in Figure 1.

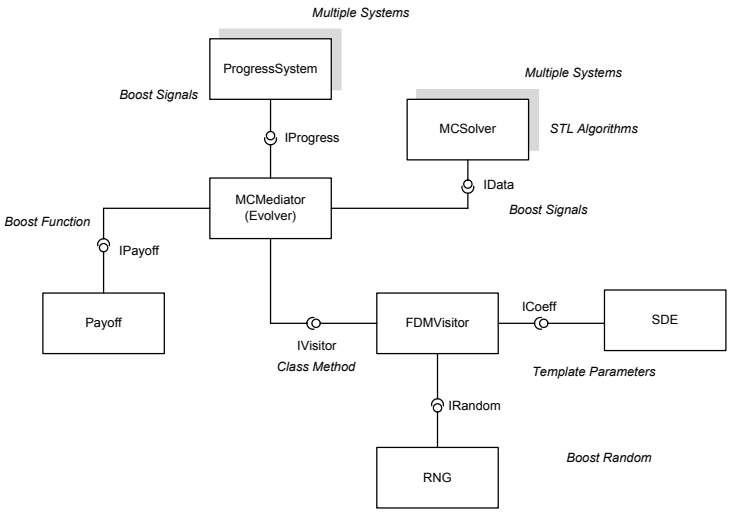

Figure 1 Context diagram for Monte Carlo application

We give a summary of the classes in this design:

- `RNG`: the classes that generate random numbers (for example, *Mersenne-Twister*).
- `SDE`: the class that models stochastic differential equations.
- `FDMVisitor`: a family of classes, each one implementing a particular finite difference method. We have used the classic GOF *Visitor* pattern.
- `Payoff`: classes that model payoffs, for example call and put options.
- `MCSolver`: the classes that implement the option pricing algorithms based on the underlying path information from the `FDMVisitor` classes.
- `MCMediator` : the central coordinator that manages control flow and data flow between the other modules in Figure 1.
- `ProgressSystem`: these are the modules that receive data from the running application, for example for statistics gathering and reporting.

In this chapter we adapt and improve the design in Figure 1 and we implement it in C#.

3. Domain Architecture and System Design

We have examined Monte Carlo option pricing from an object-oriented approach based on design patterns in C++ in Duffy and Kienitz 2009. The essential features of the design consisted of the use of classes and class hierarchies to model mathematical concepts and the application of design patterns to help promote the extendibility of the application. In this chapter, we take a different approach by decomposing the system into loosely-coupled and cohesive components having well-defined interfaces. The discovery of these components is sometimes by trial and error and this can be a time-consuming process. We do not discuss this process here but instead we summarise these ad-hoc efforts by realising that the current application is a special case of a *Resource Allocation and Tracking (RAT)* domain category as discussed in Duffy 2004. The systems in this category share the common characteristic that they process some kind of a request and produce a result relating to the status of the request. The best example of a RAT instance is a helpdesk system. The input is a user request and the output is a report (or several reports) describing the status of the request in time and space. For example, a user has placed an order to purchase a book online and she would like to know how long it will take to arrive on the doorstep. In the same vein, we see the Monte Carlo engine as having a similar structure and data flow:

1. Determine the payoff and kind of option to be priced.
2. Choose the SDE that models the behaviour of the underlying asset.
3. Determine how the SDE is approximated by using a finite difference scheme.
4. Configure system parameters and define the management system that stores the audit/performance trail data of the running engine.
5. Define the option pricers; for example, it is possible to configure the system to price several kinds of options.
6. Configure the object network using the *Builder* pattern.

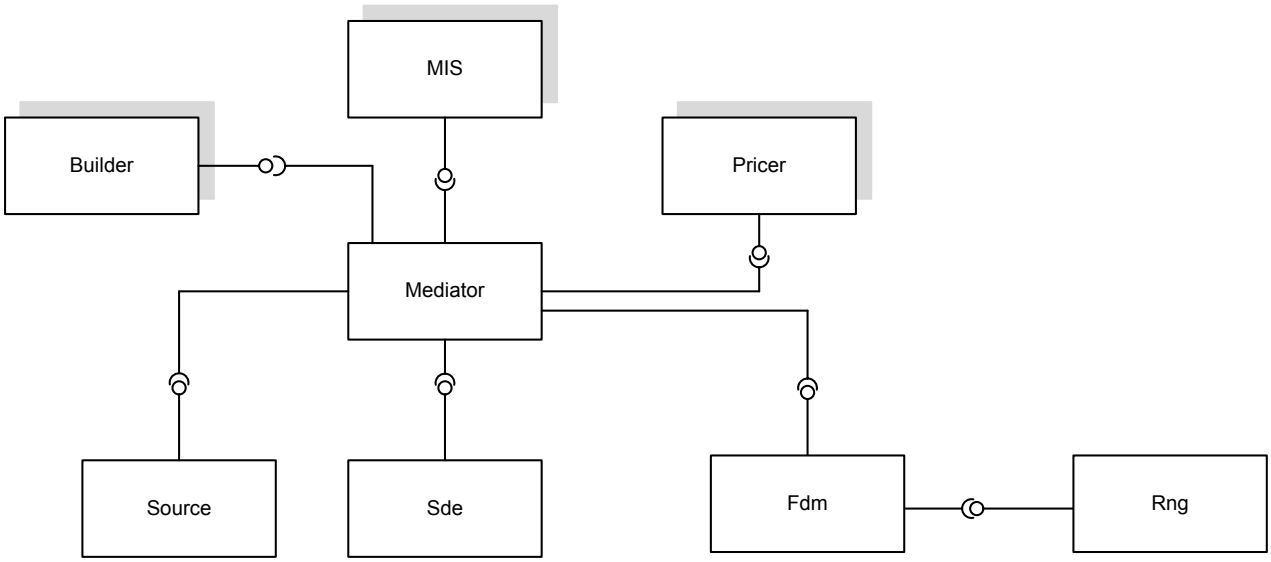

Figure 2 New Context diagram for Monte Carlo engine

The context diagram for the current application is shown in Figure 2. It is a special case of the abstract context diagram for applications that belong to the *Resource Allocation and Tracking* (RAT) category. In general, RAT systems track requests in time and space and they produce the corresponding reports relating to the status of these requests. In this case the request is to compute the price of one-factor plain, barrier, lookback and Asian options using the Monte Carlo method. The request is processed in a series of steps to produce the final output by the modules in the UML component diagram in Figure 2:

- *Source*: The system containing the data relating to the request, for example price a plain one-factor option with given market data. It contains data that is needed by other modules in Figure 2.
- *Sde*: The system that models stochastic differential equations (SDEs). In this case we model Geometric Brownian Motion (GBM) and its variants. In particular, we are interested in modelling the drift and diffusion of some underlying variable such as the stock price or interest rate, for example
- *Fdm*: The family of finite difference schemes that approximate the sdes in the Sde system. In this case we use one-step difference schemes to advance the approximate solution from one time level to the next time level until we reach the desired solution at expiration. The finite difference schemes require the services of a module that computes random numbers and standard Gaussian variates, that is variates with mean zero and standard deviation one.
- *Pricer*: This system contains classes to price one-factor options using the Monte Carlo simulation technique. The classes process path information from the *Mediator* and each class processes this path information in its own way. For example, for a plain option the pricer uses the path data at expiration, uses it to compute the payoff, adds the result to a running total and then discounts the result to compute the option price.
- *MIS*: This is the statistics-gathering system that receives status information concerning the progress of computation. For example, this system could display how many paths have been processed at any given time.
- *Builder*: This system implements a configuration/creational pattern (based on GOF 1995 but more general) that creates and initialises the systems and their structural relationships in Figure 2. The newly-created objects are encapsulated in .NET tuples which adds to the overall maintainability of the system.
- *Mediator*: This is the central coordinating entity that manages the data flow and control flow in the system. It is the driver of the system as it were and it contains the state machine that computes the paths of the SDE. It also informs the other systems of changes that they need to know about. It also plays the role of *client* in the Builder pattern (GOF 1995).
- *Rng*: a system to generate random numbers.

Having determined the global data flow in the system and the subsystems (each one having a single well-defined responsibility) we then need to determine the required-provided interfaces. These will become clear by studying the mathematics corresponding to the various components and then mapping the mathematical

functionality to C# interfaces. This will be the subject of the coming sections. For example, the C# interface to describe the one-step finite difference schemes that we use is:

```
public interface IFdm
{ // Interface for one-step FDM methods for SDEs

      // Choose which SDE model to use
      ISde StochasticEquation
      {
         get;
         set;
      }

      // Advance solution from time level t[n] to time level t[n+1]
      double advance(double xn, double tn, double dt,
            double WienerIncrement, double WienerIncrement2);
}
```

We specify all the interfaces in Figure 2 which we shall do as we progress in this chapter.
We note that the design in Figure 2 is language-independent in the sense that it can be implemented in any language that supports classes and interfaces.
We now discuss how we design the modules from Figure 2 in C#. There are several design choices and we discuss them before selecting the most appropriate one.

4. Stochastic Differential Equations (SDEs)

We are interested in modelling one-factor SDEs of the form (see Kloeden, Platen and Schurz 1997):

$$dX = \mu(t, X)dt + \sigma(t, X)dW$$

where

$$\begin{aligned} &X \equiv X_t \equiv X(t) \text{ (stochastic process)} \\ &\text{and} \\ &\mu(t,x) \in \mathbb{R} \text{ (drift)} \\ &\sigma(t,x) \in \mathbb{R} \text{ (diffusion)} \\ &W \equiv W_t \equiv W(t) \text{ (Wiener process).} \end{aligned} \tag{1}$$

In general, an SDE is uniquely specified if we define the drift, diffusion, Wiener increment and the initial condition of the stochastic process $X(t)$. Using the traditional object-oriented model this would entail creating classes or class hierarchies to encapsulate the above information. In this chapter we take a different viewpoint by focusing on the minimal set of abstract services that all sdes must deliver to clients:

- The drift function.
- The diffusion function.
- The interval $[0, T]$ on which the sde is defined.
- The initial value $X(0)$ of the stochastic process X.

In this chapter we model the above related group of characteristics in an interface. The interface contains abstract methods and properties (notice that we have changed the order of the parameters compared to those in equation (1)):

```
public interface ISde
{ // Standard one-factor SDE dX = a(X,t)dt + b(X,t)dW, X(0)given
  //                  dX = mu(X,t)dt + sig(X,t)dW


  double Drift(double x, double t);              // a (mu)
  double Diffusion(double x, double t);          // b (sig)

  // Some extra functions associated with the SDE
```

```
    double DriftCorrected(double x, double t, double B);
    double DiffusionDerivative(double x, double t);

    // Property to set/get initial condition
    double InitialCondition
    {
        get;
        set;
    }

    // Property to set/get time T
    double Expiry
    {
        get;
        set;
    }
}
```

The above methods and properties are abstract. They must be given a body in all classes that implement this interface. In other words, we create a class each time we wish to define specialised behaviour
As first example, we examine the SDE that models Geometric Brownian Motion (GBM):

$$dX_t = aX_t dt + bX_t W_t \quad a, b \text{ constant}$$
$$W_t = W(t) \text{ (one-dimensional Brownian motion)} \tag{2}$$

or more generally

$$dS_t = \mu(t)S_t dt + \sigma(t)S_t dW_t$$

where

$\mu(t)$ is the drift coefficient
$\sigma(t)$ is the diffusion coefficient. (3)

In the case of constant drift and diffusion the exact solution of (3) is given by:

$$S_t = S_0 \exp((\mu - \frac{1}{2}\sigma^2)t + \sigma W_t). \tag{4}$$

The C# class that models SDE (2) is given by:

```
public class GBM : ISde
{ // Simple SDE
  private double mu;       // Drift
  private double vol;      // Constant volatility
  private double d;        // Constant dividend yield
  private double ic;       // Initial condition
  private double exp;      // Expiry

  public GBM(double driftCoefficient, double diffusionCoefficient,
          double dividendYield, double initialCondition, double expiry)
  {
      mu = driftCoefficient;
      vol = diffusionCoefficient;
      d = dividendYield;
      ic = initialCondition;
      exp = expiry;
  }

  public double Drift(double x, double t) { return (mu - d) * x; }
```

```
    public double Diffusion(double x, double t) { return vol * x; }

    public double DriftCorrected(double x, double t, double B)
    {
        return Drift(x, t) - B * Diffusion(x, t) * DiffusionDerivative(x, t);
    }

    public double DiffusionDerivative(double x, double t)
    {
        return vol;
    }

    // Property to set/get initial condition
    public double InitialCondition
    {
        get
        {
           return ic;
        }
        set
        {
           ic = value;
        }
    }

    // Property to set/get time T
    public double Expiry
    {
        get
        {
           return exp;
        }
        set
        {
           exp = value;
        }
    }

}
```

This code is a straightforward implementation of the SDE (2) and we see that the drift and diffusion parameters are initialised by providing appropriate arguments in the constructor.
Each SDE can result in a new class and there could be some opportunity for code optimisation and refactoring in the sense that we could design a C# class that models a range of SDEs but this issue is outside the scope of this chapter. It is an optimisation step to a certain extent.

5. Numerical Approximation of SDEs

In this section we introduce the finite difference method (FDM) and we apply it to finding approximate solutions of SDEs. We define the notation that we use in this and subsequent chapters.

The first step is to replace continuous time by discrete time. To this end, we divide the interval $[0, T]$ (where T is the expiration) into a number of subintervals. We define $N + 1$ *mesh points* as follows:

$$0 = t_0 < t_1 < \ldots < t_n < t_{n+1} < \ldots < t_N = T.$$

In this case we define a set of *subintervals* (t_n, t_{n+1}) of size $\Delta t_n \equiv t_{n+1} - t_n, 0 \leq n \leq N - 1$. In general, we speak of a *non-uniform mesh* when the sizes of the sub-intervals are not necessarily the same. However we consider a large class of finite difference schemes where the N subintervals have the same size (we then speak of a *uniform mesh*), namely $\Delta t = T/N$.

We now discuss one-step finite difference schemes to approximate the solution of general one-factor SDEs. To this end, we examine the model SDE:

$$\begin{aligned} &dX(t) = \mu(X(t))dt + \sigma(X(t))dW(t) \quad 0 < t \leq T \\ &X(0) = A. \end{aligned} \tag{5}$$

We discretise the interval $[0, T]$ into N uniform subintervals and we adopt the notation:

$$\mu_n \equiv \mu(X_n), \quad \sigma_n = \sigma(X_n) \quad n = 0, .., N, \; \Delta W_n = \sqrt{\Delta t} Z, \; Z \sim N(0, 1).$$

where $N(0, 1)$ is the standard normal distribution.

Some well-known schemes are:

- Explicit Euler:
  $$X_{n+1} = X_n + \mu_n \Delta t + \sigma_n \Delta W_n. \tag{6}$$

- Semi-implicit Euler:

  with special cases

  $$\begin{aligned} &\alpha = \tfrac{1}{2} \; (\text{Trapezoidal}) \\ &\alpha = 1 \; (\text{Backward Euler}). \end{aligned} \tag{7}$$

- Heun:
  $X_{n+1} = X_n + \frac{1}{2}[F_1 + F_2]\Delta t + \frac{1}{2}[G_1 + G_2]\Delta W_n$

  where

  $$\begin{aligned} &F(x) \equiv \mu(x) - \tfrac{1}{2}\sigma'(x)\sigma(x) \\ &\text{where} \\ &; \sigma'(x) \equiv \tfrac{\partial \sigma(x)}{\partial x} \end{aligned} \tag{8}$$

  $$\begin{aligned} &F_1 = F(X_n), \quad G_1 = \sigma(X_n) \\ &F_2 = F(X_n + F_1 \Delta t + G_1 \Delta W_n) \\ &G_2 = \sigma(X_n + F_1 \Delta t + G_1 \Delta W_n). \end{aligned}$$

- Milstein:
  $$X_{n+1} = X_n + \mu_n \Delta t + \sigma_n \Delta W_n + \tfrac{1}{2}[\sigma' \sigma]_n ((\Delta W_n)^2 - \Delta t), \tag{9}$$

- Derivative-free:
  $$X_{n+1} = X_n + F_1 \Delta t + G_1 \Delta W_n + [G_2 - G_1] \Delta t^{-1/2} \tfrac{(\Delta W_n)^2 - \Delta t}{2}$$

  where

  $$\begin{aligned} &F_1 = \mu(X_n), \quad G_1 = \sigma(X_n) \\ &G_2 = \sigma(X_n + G_1 \Delta t^{1/2}). \end{aligned} \tag{10}$$

- First-order Runge Kutta with Ito coefficient (FRKI):

$$X_{n+1} = X_n + F_1\Delta t + G_2\Delta W_n + [G_2 - G_1]\Delta t^{1/2}$$

where

$$F_1 = \mu(X_n), \quad G_1 = \sigma(X_n)$$

$$G_2 = \sigma(X_n + \frac{G_1(\Delta W_n - \Delta t^{1/2})}{2}). \tag{11}$$

We now define an interface that describes how to compute the approximate solution at time level $n+1$ in terms of the known solution at time level n:

```
public interface IFdm
{ // Interface for one-step FDM methods for SDEs

  // Choose which SDE model to use
  ISde StochasticEquation
  {
      get;
      set;
  }

  // Advance solution from level t[n] to level t[n+1]
  double advance(double xn, double tn, double dt,
              double WienerIncrement, double WienerIncrement2);
}
```

We see that clients of this interface and of the classes that implement the interface must provide two Wiener increment values as input to the `advance()` method. This design decision ensures that the classes that implement the various finite difference schemes can focus on the time-marching aspects of the schemes without having to generate random numbers themselves. To this end, we have several dedicated interfaces and classes for generating random numbers that we discuss in the next section.

We now define a base class from which all specific classes that implement specific finite difference schemes are derived. It contains structure and functionality that is common to all derived classes, for example the related sde and the discrete mesh array:

```
public abstract class FdmBase : IFdm
{
  protected ISde sde;

  public int NT;           // Number of subdivisions
  public double[] x;       // The mesh array
  public double k;         // Mesh size

  public FdmBase(ISde stochasticEquation, int numSubdivisions)
  {
      sde = stochasticEquation;
      NT = numSubdivisions;
      k = sde.Expiry / (double)NT;
      x = new double[NT + 1];

      // Create the mesh array
      x[0] = 0.0;
      for (int n = 1; n < x.Length; n++)
      {
         x[n] = x[n - 1] + k;
      }
  }

  public ISde StochasticEquation
```

```
  {
      get
      {
         return sde;
      }
      set
      {
         sde = value;
      }
  }

  public abstract double advance(double xn, double tn, double dt,
                          double WienerIncrement, double WienerIncrement2);
}
```

We see a pattern emerging here: first, at the highest level we define an interface that defines the common abstract services that (unspecified) components implement. In this case we specify the methods that are common to all one-factor finite difference methods, namely defining which SDE they are approximating and the formula that advances the approximate solution from level n to level $n+1$. Secondly, we create a base class that implements the interface's methods (either by redeclaring its methods or implementing them). This base class contains the functionality common to all classes that implement finite difference schemes such as the mesh array and the step size, for example. It can also contain methods that are shared by its derived classes. In fact, this is where the application of the *Template Method* pattern (GOF 1995) is most appropriate in order to promote code reuse and to avoid costly and refactoring activities later. In short, the interface encsapsulates pure behaviour while the base class contains structural information common to all derived classes. Of course, we are referring to the *ISA* (*Gen-Spec*) relationship between classes.
The classes that implement specific finite difference schemes are reasonably lightweight and stateless in general. For example, these classes have no functionality for generating random numbers (for example) as these are created by other classes. Some classes may have their own specific member data.

As a first example, the simplest finite difference scheme is probably the explicit Euler scheme (7). The corresponding code is:

```
public class EulerFdm : FdmBase
{
  public EulerFdm(ISde stochasticEquation, int numSubdivisions)
              : base(stochasticEquation, numSubdivisions) { }

  public override double advance(double xn, double tn, double dt,
              double normalVar, double normalVar2)
  {
       return xn + sde.Drift(xn, tn) * dt
         + sde.Diffusion(xn, tn) * Math.Sqrt(dt) * normalVar;
  }
}
```

Another popular scheme due to Milstein (scheme (10)) is:

```
public class MilsteinFdm : FdmBase
{
  public MilsteinFdm(ISde stochasticEquation, int numSubdivisions)
            : base(stochasticEquation, numSubdivisions) { }

  public override double advance(double xn, double tn, double dt,
              double normalVar, double normalVar2)
  {
      return xn + sde.Drift(xn, tn) * dt
               + sde.Diffusion(xn, tn) * Math.Sqrt(dt) * normalVar
               + 0.5 * dt * sde.Diffusion(xn, tn)
               * sde.DiffusionDerivative(xn, tn) * (normalVar * (dynamic)normalVar - 1.0);
  }
```

```
}
```

As final example, we discuss the *modified predictor-corrector method*:

$$X_{n+1} = X_n + \left\{\alpha\overline{\mu}_\beta(\tilde{X}_{n+1}) + (1-\alpha)\overline{\mu}_\beta(X_n)\right\}\Delta t + \left\{\beta\sigma(\tilde{X}_{n+1}) + (1-\beta)\sigma(X_n)\right\}\Delta W_n, n \geq 0 \tag{12}$$

where the *corrector drift function* is defined by:

$$\overline{\mu}_\beta(x) = \mu(x) - \beta\sigma(x)\frac{\partial\sigma}{\partial x}(x). \tag{13}$$

Furthermore, you can customise the scheme to support different levels of implicitness and explicitness in the drift and diffusion terms:

A. Fully explicit($\alpha = \beta = 0$)
B. Fully implicit($\alpha = \beta = 1$)
C. Implicit in drift explicit in diffusion ($\alpha = 1, \beta = 0$)
D. Symmetric($\alpha = \beta = 1/2$).

The code that implements this scheme is:

```
public class ModifiedPredictorCorrectorFdm : FdmBase
{ // PC using adjusted drift

  private dynamic A, B, VMid;

  public ModifiedPredictorCorrectorFdm(ISde stochasticEquation,
                              int numSubdivisions, double a, double b)
                              : base(stochasticEquation, numSubdivisions)
  {
      A = a;
      B = b;

  }

  public override double advance(double xn, double tn, double dt,
                          double normalVar, double normalVar2)
  {

      // Euler for predictor
      VMid = xn + sde.Drift(xn, tn) * dt
                 + sde.Diffusion(xn, tn) * Math.Sqrt(dt) * normalVar;

      // Modified Trapezoidal rule
      double driftTerm = (A * sde.DriftCorrected(VMid, tn + dt, B)
                + ((1.0 - A) * sde.DriftCorrected(xn, tn, B))) * dt;

      double diffusionTerm = (B * sde.Diffusion(VMid, tn + dt)
                   + ((1.0 - B) * sde.Diffusion(xn, tn))) * Math.Sqrt(dt)
                   * normalVar;

      return xn + driftTerm + diffusionTerm;
  }
}
```

It is clear that this method is more computationally intensive than either the Euler or Milstein methods because of the greater number of function calls.

6. Random Number Generation

One of the tools that is needed in a Monte Carlo simulator is a suitable random number generator. It is not our intention to discuss this in great detail here but we do discuss some mathematical background as well as code to compute pseudo-random numbers and how this code fits into the current framework. In general we first generate uniform random numbers on the unit interval and based on these numbers we generate standard normal variates. We first describe two well-known methods for generating normal variates.

6.1 Polar Marsaglia Method

This method uses the insight that if the random variable U is $U(0,1)$ then the random variable V defined by $V = 2U - 1$ is $U(-1,1)$. We now choose two variables defined by:

$$V_j = 2U_j - 1, \quad U_j \sim U(0,1), \quad j = 1,2.$$

Then we define:

$$W = V_1^2 + V_2^2 \leq 1, \quad W \sim U(0,1).$$

We keep trying with different values until the above inequality is satisfied. Continuing, we define the intermediate value:

$$Y = \sqrt{-2\log(W)/W}.$$

Finally, the pair of values defined by:

$$N_j = V_j Y, \quad j = 1,2$$

are two standard normally (Gaussian) distributed random variables, and we are done.

6.2 Box-Muller Method

This method is based on the observation that if r and φ are two independent $U(0,1)$ random variables then the variables:

$$N_1 = \sqrt{-2\log r}\cos(2\pi\varphi)$$

$$N_2 = \sqrt{-2\log r}\sin(2\pi\varphi).$$

are two independent standard Gaussian random variables.

We now show how we have programmed these methods in C#. We have a number of choices for generating uniform numbers:

- Using the .NET `Random` class.
- Using the .NET `RandomNumberGenerator`.
- Using an external C++ library that we can use from C# by means of the *C++/CLI* language. This solution is outside the scope of this chapter.

To this end, we first discuss the design of the subsystem for generating random numbers. What do we need? The answer is a random number. The top-level specifications are:

```
public interface IRng
{
  double GenerateRn();
}

public abstract class Rng : IRng
{
  public abstract double GenerateRn();
}
```

All concrete classes for generating random numbers are derived from `Rng` , two of which are:

```
public class PolarMarsagliaNet : Rng
{
  private Random rand;

  public PolarMarsagliaNet() {  rand = new Random();}

  public override double GenerateRn()
  {

      double u, v, S;

      do
      {
         u = 2.0 * rand.NextDouble() - 1.0;
         v = 2.0 * rand.NextDouble() - 1.0;
         S = u * u + v * v;
      }
      while (S >= 1.0);

      double fac = Math.Sqrt(-2.0 * Math.Log(S) / S);
      return u * fac;
  }
}
```

and

```
public class BoxMullerNet : Rng
{
  private Random rand;

  public BoxMullerNet() { rand = new Random(); } // Seed is from system clock
  double U1, U2;

  public override double GenerateRn()
  {
      // U1 and U2 should be independent uniform random numbers
      U1 = rand.NextDouble();   // In interval [0,1)
      U2 = rand.NextDouble();  // In interval [0,1)

      // Box-Muller method
      return Math.Sqrt(-2.0 * Math.Log(U1)) * Math.Cos(2.0 * 3.1415159 * U2);
  }
}
```

These classes generate normal variates for the Polar Marsaglia and Box-Muller method, respectively. In both cases we use the .NET `Random` class to generate the related uniform random numbers.

Finally, we can avoid code duplication in the above classes by using *plug-in methods*. Thus, instead of hard-coded uniform random number generators we use a delegate with the appropriate signature:

```
public delegate double RngDelegate();
```

The class with an embedded delegate is:

```
public class BoxMullerII : Rng
{ // RNG with embedded delegate

  private RngDelegate rand;
  private double U1, U2;

  public BoxMullerII(RngDelegate randomGenerator)
```

```
  { rand = randomGenerator; }

  public override double GenerateRn()
  {
      // U1 and U2 should be independent uniform random numbers
      U1 = rand();          // In interval [0,1)
      U2 = rand();          // In interval [0,1)

      // Box-Muller method
      return Math.Sqrt(-2.0 * Math.Log(U1)) * Math.Cos(2.0 * 3.1415159 * U2);
  }
}
```

The advantage of this approach is that we can use the class with any method that has the same signature as that of `RngDelegate`, for example:

```
// RNG using delegates
Random rngNet = new Random();
RngDelegate rand = rngNet.NextDouble;
IRng rng = new BoxMullerII(rand);
Console.WriteLine("Rng via delegate {0}", rng.GenerateRn());
```

7. Pricers

We now describe the component to price one-factor options using the Monte Carlo method. In this section we focus on plain options in order to motivate the software design. The interface specification is:

```
public interface IPricer
{
  void ProcessPath(ref double[] arr);     // The path from the evolver
  void PostProcess();                     // Finish off computations
  double DiscountFactor();                // (simple) discounting function
  double Price();                         // Computed option price
}
```

At the next level we define an abstract class:

```
// The payoff function
public delegate double Payoff(double underlying);

public abstract class Pricer : IPricer
{
  public abstract void ProcessPath(ref double[] arr);     // Create a single path
  public abstract void PostProcess();                     // Notify end of simulation
  public abstract double DiscountFactor();                // Discounting
  public abstract double Price();                         // Option price

  public Payoff m_payoff;
  protected Func<double> m_discounter;

  public Pricer(Payoff payoff, Func<double> discounter)
  {
      m_payoff = payoff;
      m_discounter = discounter;
  }
}
```

The code for a plain one-factor pricer is:

```
// Pricing Engines
public class EuropeanPricer : Pricer
{
  private dynamic price;
```

```
    private dynamic sum;
    private int NSim;

    public EuropeanPricer(Payoff payoff, Func<double> discounter)
           : base(payoff, discounter) { price = sum = 0.0; NSim = 0; }

    public override void ProcessPath(ref double[] arr)
    { // A path for each simulation/draw

        // Sum of option values at terminal time T
        sum += m_payoff(arr[arr.Length - 1]); NSim++;
    }

    public override double DiscountFactor()
    { // Discounting

        return m_discounter();
    }

    public override void PostProcess()
    {
        Console.WriteLine("Compute Plain price");   price = DiscountFactor() * sum / NSim;
        Console.WriteLine("Price: {0}, {1}", price, NSim);
    }

    public override double Price()
    {
        return price;
    }

}
```

8. System Configuration and Interface Specification

Having documented, designed and implemented the software modules and their interfaces from Figure 2 we must now decide how to instantiate them. Specifically, we decide which specific classes will implement these interfaces. We instantiate the classes and then we add them to the end-product which is the network object in Figure 2. For example, we may wish to price a barrier option based on GBM using the Euler method. Furthermore, we may choose to generate random numbers using the Box-Muller method in combination with the .NET `Random` class. To this end, we must instantiate the appropriate classes and configure the software system with these choices. More generally, we need to execute the following steps:

1. Initialise each of the modules and their data in Figure 2.
2. Connect the modules based on the *provides-requires* model. It is at this stage that we decide whether to model data flow using events (*push model*) or by methods (*pull model*).
3. Start the application.

Since the system is structured as a collection of cohesive and loosely-coupled subsystems we see that it is relatively easy to configure it. We may need to introduce a number of new classes and functions that allow the system to communicate with external hardware and software systems, for example:

1. Data sources containing settings, default values and user preferences (for example, databases, text files, user interfaces such as the console and graphical user interfaces).
2. Hardware drivers such as assemblies and dlls. For example, we can encapsulate a C++ random number generator that we stored in an assembly and that then can be loaded into memory at run-time and whose functionality can used by the C# code.

In the interest of completeness, we should document the emergence of these new low-level classes by extending Figure 2 to form a more detailed design-level system context diagram. We see the emergence of new layers.

One of the objectives of this chapter is to create a customisable software system to price one-factor options using the Monte Carlo method. The method should support a range of SDEs, finite difference methods and

random number generators. These are the abstractions that we create in the *Builder* pattern that we use to configure the most important modules in Figure 2. We have not included the pricer classes in this builder because doing so would make it less reusable. Instead, the responsibility for their creation takes places elsewhere. This tactic also avoids our having to create and maintain an unwieldy *mega-builder*. To this end, we use a generic delegate to specify the interface for the builder:

```
// Generic delegate for a MC builder: T1 == Sde, T2 == Fdm, T3 == IRng
public delegate Tuple<T1, T2, T3> Builder<T1, T2, T3>();
```

We see that the delegate has three generic parameters that will be instantiated as shown in the above commented line. In particular, we design a builder class that creates the SDE, FDM and RNG components in Figure 2. It uses the Console to elicit input from the user. Furthermore, the builder uses and needs the parameters corresponding to well-known SDEs. The builder class' parameters have generic constraints defined on them; furthermore, this class has methods for creating the SDE, FDM and RNG instances as well as a factory method to return all three parts (the product) that conforms to the signature of the above delegate type:

```
public class MCBuilder<S, F, R>
  where S : ISde
  where F : IFdm
  where R : IRng
{ // Build the full UML model in this builder

  // Next version .. encapsulate better
  private double r;
  private double v;
  private double d;
  private double IC;
  private double T;
  private double beta;

  // Constructor (data important at this stage)
  public MCBuilder(Tuple<double, double, double, double, double, double> data)
  {
      r = data.Item1;
      v = data.Item2;
      d = data.Item3;
      IC = data.Item4;
      T = data.Item5;
      beta = data.Item6;
  }

  public Tuple<S, F, R> Parts(S sde, F fdm, R rng)
  { // V1, parts initialised from the outside

      return new Tuple<S, F, R>(sde, fdm, rng);
  }

  public Tuple<ISde, FdmBase, IRng> Parts()
  { // V2, parts initialised from the inside

      // Get the SDE
      ISde sde = GetSde();
      IRng rng = GetRng();
      FdmBase fdm = GetFdm(sde);
      return new Tuple<ISde, FdmBase, IRng>(sde, fdm, rng);
  }

  private ISde GetSde()
  {
      Console.WriteLine("Create SDE");
      Console.Write("1. GBM, 2. CEV ");
      int c = Convert.ToInt32(Console.ReadLine());
```

```
    if (c == 1)
    { // GBM

       return new GBM(r, v, d, IC, T);
    }
    else
    {
       return new CEV(r, v, d, IC, T, beta);
    }
}

private IRng GetRng()
{
    Console.WriteLine("Create RNG");
    Console.WriteLine("1. Box-Muller .Net 2. Polar Marsaglia Sitmo”);
    // more …
    int c = Convert.ToInt32(Console.ReadLine());

    IRng rng;

    switch (c)
    {
       case 1:
          rng = new BoxMullerNet();
          break;
       case 2:
          rng = new PolarMarsagliaSitmo();
          break;

    // more …

    default:
       rng = new BoxMullerSitmo();
       break;
    }

    return rng;
}

private FdmBase GetFdm(ISde sde)
{
    Console.WriteLine("Create FDM");
    Console.WriteLine("1. Euler, 2. Extrapolated Euler, 3. Milstein ");
    // more
    int c = Convert.ToInt32(Console.ReadLine());

    FdmBase fdm;

    int NT = 500;
    Console.Write("How many NT? ");
    NT = Convert.ToInt32(Console.ReadLine());

    double a, b;

    switch(c)
    {
       case 1:

          fdm = new EulerFdm(sde, NT);
          break;
    case 2:
```

```
            fdm = new ExtrapolatedEulerFdm(sde, NT);
            break;

      case 3:

            fdm = new MilsteinFdm(sde, NT);
            break;

      // more…
         default:
            fdm = new ExtrapolatedEulerFdm(sde, NT);
            break;
      }

      return fdm;
  }
}

  public static Builder<ISde, FdmBase, IRng> ChooseBuilder(int n)
  {// Factory method to choose your builder

      double r = 0.08;
      double v = 0.3;
      double div = 0.0;
      double IC = 60.0;
      double T = 0.25;
      double K = 65.0;
      double beta = 1.0;

      Tuple<double, double, double, double, double, double> data = new Tuple<double,
      double, double, double, double, double>
            (r, v, div, IC, T, beta);

      MCBuilder<ISde, FdmBase, IRng> builder = new MCBuilder<ISde, FdmBase, IRng>(data);

      return builder.Parts;
  }
```

This builder class is not as general as we would like and in future versions we could have different kinds of builders for different kinds of users. To this end, we create a *factory method* that allows us to choose a suitable builder based on a decision and we have encapsulated this decision in a method:

```
// Choose which builder you want
public static Tuple<ISde, FdmBase, IRng> ChooseBuilder(int n)
{// Factory method to choose your builder

double r = 0.08;
double v = 0.3;
double div = 0.0;
double IC = 60.0;
double T = 0.25;
double K = 65.0;
double beta = 1.0;

MCBuilder<ISde, FdmBase, IRng> builder;
MCDefaultBuilder<ISde, FdmBase, IRng> builder2;

int c = 1;
Console.Write("1. MCBuilder, 2. Default Builder ");
c = Convert.ToInt32(Console.ReadLine());

if (1 == c)
```

```
{
  Console.Write("Chosen 1. MCBuilder ");
  Tuple<double, double, double, double, double, double> data = new
            Tuple<double, double, double, double, double, double>
            (r, v, div, IC, T, beta);
  builder = new MCBuilder<ISde, FdmBase, IRng>(data);
  return builder.Parts();
}
else
{
Console.Write("Chosen 2. MCDefaultBuilder ");
builder2 = new MCDefaultBuilder<ISde, FdmBase, IRng>();
return builder2.Parts();
}

}
```

We can then call this method in the `Main()` method which adds to the readability and maintainability of the code:

```
Tuple<ISde, FdmBase, IRng> factory = ChooseBuilder(choice);
```

In `Main()` the product that the builder creates is given by the following code:

```
// Choose which builder to use
int choice = 1;
Tuple<ISde, FdmBase, IRng> parts2 = ChooseBuilder(choice);
```

9. Putting it All Together: The *Mediator*

Having designed the software components that we have already discussed in this chapter we need to assemble them to form a working system. This process is particularly easy in the current case because we have a library of loosely-coupled and cohesive components that will be managed by a planning and coordination component called a *mediator*. Instead of assembling the components in an ad-hoc manner we see that the mediator becomes an explicit component in the overall software architecture. Mediators have sufficient semantic complexity and runtime autonomy (persistence) and these properties allow them to play the role of first-class entities in a software architecture (Bass, Clemens and Kazman 1998). The *Mediator* pattern is described in GOF 1995. We note some of the attention points to be addressed:

1. Create the components in Figure 2 that the mediator needs (these components are created by a builder).
2. Determine the *provides/requires* interfaces between the mediator and the other components.
3. Design the data flow, control flow and state machine associated with the mediator.

The mediator's main responsibility is to coordinate the other components in the system. In general it contains no code for object creation nor does it communicate with object factories. It receives all its component via its constructor. This improves its maintainability.

We design the mediator class to reflect the component diagram in Figure 2. It has a constructor that accepts a tuple containing the components that it needs and it has two public events that define the communication with the pricer components (we shall see how to assign these events in the next section):

```
// Events
public delegate void PathEvent<T> (ref T[] path);        // Send a path array
public delegate void EndOfSimulation<T>();               // No more paths
```

Finally, the mediator has a method called `start()` that is responsible for path generation:

```
public class MCMediator
{
  // Three main components
  private ISde sde;
  private FdmBase fdm;
  private IRng rng;
```

```
    // Other MC-related data
    private int NSim;                                // Number of simulations
    private double[] res;                            // Generated path per simulation

    // Event notification
    public event PathEvent<double> path;             // Signal to the Pricers
    public event EndOfSimulation<double> finish;     // All paths are complete

    public MCMediator(Tuple<ISde, FdmBase, IRng> parts, int numberSimulations)
    {
        sde = parts.Item1;
        fdm = parts.Item2;
        rng = parts.Item3;

        NSim = numberSimulations;
        res = new double[fdm.NT];
    }

    public void start()
    { // Main event loop for path generation

        double VOld, VNew;

        for (long i = 1; i <= NSim; ++i)
        { // Calculate a path at each iteration

           if ((i / 5000) * 5000 == i)
           { // Give status after a given numbers of iterations

              Console.WriteLine(i);
           }

           VOld = sde.InitialCondition; res[0] = VOld;

           for (int n = 1; n < res.Length; n++)
           { // Compute the solution at level n+1

              VNew = fdm.advance(VOld, fdm.x[n-1], fdm.k,
                     rng.GenerateRn(), rng.GenerateRn());
              res[n] = VNew; VOld = VNew;
           }

           // Send path data to the Pricers
           path(ref res);
        }
        finish();     // Signal to pricers to finish up

    }
}
```

The main advantage is that the *Mediator* pattern encapsulates how a set of components interact. It promotes loose coupling by keeping components from explicitly referring to each other.

9.2 The Relationship with Bridge and Adapter (Wrapper) Patterns

The mediator is the central component in Figure 2. It communicates with other components using *provides/requires* interfaces. The components implementing these interfaces may have multiples layers of indirection and we should try to flag as many assumptions as possible that components make about their environment. In general, we wish to avoid or mitigate *interface mismatch* between software layers and components. We use the components in a variety of different but currently unknown contexts. It is important to design with change in mind and to adopt a disciplined approach from the earliest design stages. To this end, we can use two patterns that complement *Mediator* (see GOF 1995):

- *Bridge*: we decouple a component from its implementation so that the two can vary independently. Bridges address specific issues such as translation between different protocols and more generally by promoting information hiding and enhancing the resultant inter-component interoperability. Bridges are needed when we wish to support legacy software.
- *Adapter* (*Wrapper*): an adapter is a component that converts the interface of a component into another interface that clients expect. Adapters let components work together that would not normally be able to communicate because of incompatible interfaces.

We give a number of exercises in this chapter that discuss how to apply these patterns to the Monte Carlo simulator.

10. Examples and Test Cases

We now take a test case to show how to price a number of options and show how the components work together. In general, the following steps are executed and they can be seen as forming a pattern for other kinds of applications:

- Initialise the option data:

```
double r = 0.08;
double v = 0.3;
double div = 0.0;
double IC = 60.0;
double T = 0.25;
double K = 65.0;
double beta = 1.0;
```

- Choose the builder that creates the components in Figure 2 as discussed in section 8:

```
// Choose which builder to use
int choice = 1;
Tuple<ISde, FdmBase, IRng> parts2 = ChooseBuilder(choice);
```

- Create the mediator (central component in the application):

```
int NSim = 1000000;
Console.Write("How many NSim? ");
NSim = Convert.ToInt32(Console.ReadLine());

MCMediator mcp = new MCMediator(parts2, NSim);
```

- Create the payoff functions:

```
// Use lambda functions to define payoffs and discounting
Payoff payoff = x => Math.Max(0.0, K - x);
//  Payoff payoff = x => Math.Max(0.0, x - K);

Func<double> discounter = () => Math.Exp(-r * T);
```

- Create prices and link them to the mediator so that they become the recipients of events from the mediator (notice the use of multicast delegates:

```
// Manually create pricers
IPricer op = new EuropeanPricer(payoff, discounter);
IPricer op2 = new BarrierPricer(payoff, discounter);
IPricer op3 = new AsianPricer(payoff, discounter);

// Define slots for path information
mcp.path += op.ProcessPath;
mcp.path += op2.ProcessPath;
mcp.path += op3.ProcessPath;
mcp.path += op4.ProcessPath;
```

```
// Signal end of simulation
mcp.finish += op.PostProcess;
mcp.finish += op2.PostProcess;
mcp.finish += op3.PostProcess;
mcp.finish += op4.PostProcess;
```

- Run the program and examine the output:

```
// Create and start the stopwatch
Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();

mcp.start();
stopWatch.Stop();
// Get the elapsed time as a TimeSpan value.
TimeSpan ts = stopWatch.Elapsed;

// Format and display the TimeSpan value.
string elapsedTime = String.Format("Elapsed time {0:00}:{1:00}:{2:00}.{3:00}",
                    ts.Hours, ts.Minutes, ts.Seconds, ts.Milliseconds / 10);
Console.WriteLine(elapsedTime, "RunTime");
```

We are finished! This code can form the basis for more general solutions.

11. The Project Management and Software Architecture Perspectives

Figure 2 represents the architecture of the software system that we have discussed in this chapter. It did not originate as a *big bang* process as it were but it is a special case of a domain-independent model that the author has developed and applied to create various kinds of applications (see Duffy 2004). Furthermore, prior to having developed the solution to Figure 2 we had already written a similar prototype in C++ to price options using the Monte Carlo Method. We can conclude that the system is reasonably stable and we use it as a common reference model that all the members of the software team use in order to communicate with each other. In this way we hope to bridge the communication gap between the different project stakeholders. Most software projects have budget and time-to-market constraints. We do not discuss what the roles of these stakeholder groups are here but we do give some guidelines on how they can use and integrate the software assets from Figure 2 into their work practices.

We first discuss some issues relating to the software architecture in Figure 2 and how it is updated and modified during the lifetime of the software organisation that supports it. We can set out some goals and objectives

1. The software architecture is a living organism in the sense that it evolves and improves as new functionality is added to it. All stakeholders can use it as their *reference model*.
2. Many of the components in the software architecture are *reusable assets* in the sense that they can be used in other applications and not just in the one under discussion. For example, the components for random number generation and payoffs are certainly reusable software components.
3. The software architecture in Figure 2 (being an instance application of a RAT category) can be used as a template for other applications in the same category. For example, the current architecture can be morphed to produce an architecture for a software system to price one-factor options using partial differential equations (PDE) and the finite difference method (FDM). In general, points 2 and 3 correspond to "*communitywide reuse of architectural assets*" as discussed in Bass, Clements and Kazman 1998. In particular, the Monte Carlo system in an *exemplar system* that can be used as a demonstration type.
4. This point is a follow-on from points 2 and 3. Now we are interested in *component-based product lines* representing *program families* that we create using reusable components and architectures. A product line is a collection of systems sharing a managed set of features constructed from a common set of core software assets. These assets include a base architecture and a set of common and possibly tailorable components that populate it.

Closely related to this discussion is the challenge of estimating the effort (manhours) needed in order to create new software assets, integrating them into the architecture and deploying them in applications. Determining

the number of hours to create a component can be estimated using the following formula from *PERT* (*Project Estimation and Review Technique*). To use the formula we need three estimates (*uncertainty parameters*) :

- Parameter a (most optimistic or shortest time to complete the task).
- Parameter b (most likely or model time).
- Parameter c (most pessimistic or longest time).

The values of these parameters should be estimated using a combination of historical data, experience and by interviewing experts. We model the estimation process by the *triangular probability distribution* with parameters a, b and c. Finally, a statistical estimate of the expected time is given by the following weighted average:

$$T = (a + 4b + c)/6.$$

We apply this formula to each activity in the project.
Everyone in the team can use this formula as a rough estimate. It is in any case better than ad-hoc guessing!

12. Advantages and Benefits

Having gone to the effort of designing and implementing a flexible Monte Carlo engine we can ask ourselves what the advantages are. A general answer is that the approach is based on system decomposition, interfaces and delegates. This results in software that is easier to adapt and even reuse than solutions based on the procedural and object-oriented programming models. Some alternative solutions are:

a) *Procedural model* (Clewlow and Strickland 1998, Webber 2011): this style basically entails writing the software by creating a double loop that iterates over the number of simulations (outer loop) and over the number of time steps to approximate the SDE path (inner loop). The financial logic is implemented inside the inner loop.
b) *Object-oriented model* (Duffy and Kienitz 2009, Webber 2011): this is an attempt to adapt the software in step a). We use the standard modelling techniques such as encapsulation, composition and inheritance to promote the adaptability of the resulting software system. Maintainability and understandability become the main challenges with this approach.

These solutions can break down when we wish to extend the software to suit new requirements, for example:

- Extending the software to support different kinds of products and models.
- Achieving independence between models, products, market data and model parameters.
- Interoperability with other software and hardware systems.

In order to achieve these ends we can use a combination of modern language features (such as interfaces and delegates) in combination with system architectures and patterns as described in Duffy 2004 and POSA 1996. In particular, the *Layers* pattern is widely-used to create virtual machines and to realise an extra level of indirection between software modules. An example of what we mean (calibration of SDEs) is discussed in exercise 2 of this chapter.

13. Generalisations and Applications

The architecture in Figure 2 is based on the generic RAT model from Duffy 2004 where we discuss the rationale for the model and we also give some examples and applications. In the context of computational finance it is relatively easy to adapt the model to suit other pricing problems, for example, lattice (binomial) and PDE/FDM models. When commencing on such an *analogical reasoning project*, the following general issues should be borne in mind (Polya 1957):

- Understand the problem: what is the output from the system and what is the input?
- Devise a plan: find the connection between the input and output. In the current context, determine how output is created from input by a collection of loosely-coupled and cohesive subsystems as we saw in Figure 2. Create the system context diagram and determine what you wish to implement and do not wish to implement.
- Carry out the plan: design and implement the solution in C#. You will not be able to implement all requirements in one iteration which means that the final solution will be the end product of a series of prototypes with ever-increasing functionality.

- Looking back: examine the solution and determine if it satisfies the current requirements. Additionally, can you reuse the design or code in other systems and applications? For example, it might be possible to use mesh generators, finite difference schemes, random number generators and payoff classes in other applications.

These work practices demand that you take (and get!) the time to create software as a series of prototypes. This may not be possible due to organisational and political reasons. A discussion of these issues is outside the scope of this book.

14. Summary and Conclusions

In this chapter we discussed the design and implementation of a software system to price one-factor option models using the Monte Carlo method. We employed a combination of design techniques that have been known, the author's *Domain Architectures* and modern language features in the .NET Framework and C#.
We discussed the problem by approaching the problem in a well-defined manner, starting by describing the financial problem and the closely related mathematical model. We then approximated this model using finite difference methods. Finally, we implemented these models.
We recommend that you study and do the exercises in this chapter as they include discussions on extending the UML component diagram in Figure 2 to suit new requirements.

15. Exercises and Projects

1. (Interfaces versus delegates; interfaces with delegates)

In section 4 we discussed how to implement SDEs in C# using specialisation (subtype polymorphism). This is the traditional object-oriented approach. Some of the disadvantages are:

a) Inflexible and difficult-to-maintain class hierarchies.
b) In some cases we model roles as classes when we should model them as instances of classes to allow run-time switching.
c) Configuring systems that use classes from class hierarchies adds to complexity.

The problem here is that the coupling between the classes and their methods is hard-wired. We employ delegates in order to add an extra level of indirection between classes and methods. In this sense we realise that we have a candidate example of the *Bridge* pattern. Answer the following questions:

a) Create a class to model an SDE. It has member data for initial condition and expiry. It also implements the drift and diffusion functions using delegates.
b) What are the advantages of this approach compared with the object-oriented approach in section 4? Do we need a class hierarchy?
c) Now consider the third solution as shown in Figure 3 which can be considered as a combination of the object-oriented solution and the solution in part a) of this exercise. In particular, we implement the drift and diffusion functions by delegates and the SDE initial condition and expiry by member data in a single class `Sde`.
d) What are the advantages of the approach in part c) compared to the object-oriented solution and the solution in part a) of this exercise?

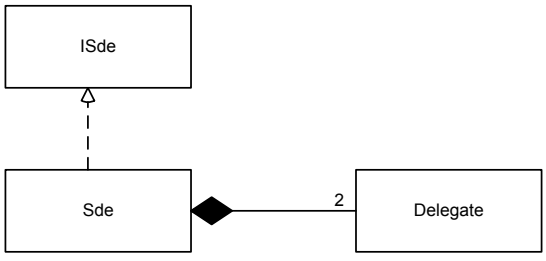

Figure 3 Alternative solution

2. (Calibration of SDEs mini-project)

The SDEs that we discussed in this chapter assume that we have an exact or analytic representation for the drift and diffusion functions in the SDE. In many cases however, these functions are not known exactly because

they depend on unknown parameters that must be estimated in some way. In this case we write the SDE to show dependence on a parameter θ (this is usually a vector):

$$\begin{cases} dX_t = a(X_t, t; \theta)dt + b(X_t, t; \theta)dW_t, t \geq 0 \\ \\ X_0 = x_0. \end{cases}$$

We assume that the initial value x_0 is deterministic and that $x_0, x_1, \ldots, x_n$ is a sequence of historical observations from the above stochastic process that is sampled at the deterministic discrete time-points $t_0 < t_1 < \ldots < t_n$. For example, the one-factor *Ornstein-Uhlenbeck* (OU) process depends on three unknown parameters λ, μ and σ:

$$\begin{cases} dS_t = \lambda(\mu - S_t)dt + \sigma dW_t \\ \\ \text{where} \\ \\ \lambda = \text{mean reversion rate} \\ \mu = \text{mean} \\ \sigma = \text{volatility}. \end{cases}$$

The process can be considered as a modification of the random walk in continuous time, or Wiener process, in which the properties of the process have been changed so that there is a tendency of the walk to move back towards a central location, with a greater attraction when the process is further away from the centre. The Ornstein–Uhlenbeck process can also be considered as the continuous-time analogue of the discrete-time autoregressive AR(1) process.

There are several techniques to estimate the parameters in these SDEs., some of which are:

i) *Maximum Likelihood Estimator* (MLE) of the unknown parameters can be calculated if the transition density of the stochastic process is known.
ii) Solve numerically the *Kolmogorov partial differential equation* satisfied by the transition density when the transition density is not known.
iii) *Least Squares Regression* (LSR) in which we use time-series data or simulated data to find the unknown parameters. For example, for the OU process we assume the AR(1) form:

$$S_{i+1} = aS_i + b + \varepsilon \quad (\varepsilon \text{ is } \quad \text{i.i.d. } N(0,1)).$$

We use a linear least squares algorithm to find the parameters a and b which then allow us to determine the parameters of the process.

Answer the following questions:

a) Determine how to accommodate the new requirement that the drift and diffusion functions can be computed in different ways. In other words, consider which of the solutions in exercise 1 will allow us to meet these requirements. In fact, you need to implement the *Bridge* pattern as shown in Figure 4.

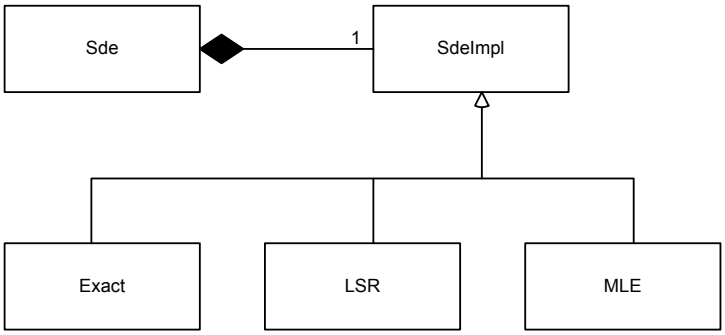

Figure 4 Extra level of Indirection

b) Update the system context diagram in Figure 2.
c) Implement classes that estimate the parameters in the above SDEs. Test you new code on the OU process that model the Vasicek model using exact simulated data.
d) It may be useful to apply the *Layers* pattern to design and implement this problem. Pay particular attention to the following steps (as discussed in POSA 1996):
   i) The number of layers needed.
   ii) The services that a layer delivers to its upper layers.
   iii) Specify the interfaces for each layer.
   iv) Structure each layer (for example, design the classes that implement each layer).
   v) Decouple adjacent layers.
   vi) Design an error-handling strategy.
e) Concerning data and control flow between the different layers, determine when you would use a *push model* using events or a *pull model* using methods.

3. (What would we do if we did not have a mediator?)

Discuss the (many) advantages of using the *Mediator* patterns in your designs.

In particular, answer the following questions:

a) Consider the system architecture in Figure 2 if the mediator component were not present. What are the consequences for reusability and maintainability?
b) Discuss how the *Mediator* pattern can help performance by caching data; some computations only need to be executed once but at the expense of extra memory consumption.
c) The ability to use various mediator classes to suit different needs. Which factory patterns would you use to create these various mediators?
d) Discuss the following advantages and disadvantages of using a mediator:
   - It limits subclassing.
   - It decouples the participating components.
   - It simplifies object protocols.
   - It abstracts how objects cooperate.
   - It centralises control; mediators can become top-heavy.

4. (Analogous Reasoning: from Monte Carlo Method to PDE Models)

The objective of this extended exercise is to take an existing design and implementation of a Monte Carlo model to price one-factor options and adapt it so that its architecture becomes similar to that in Figure 2. As preparation, you need to know the design rationale for the architecture in Figure 2 as well as the code base for the model in Figure 5.

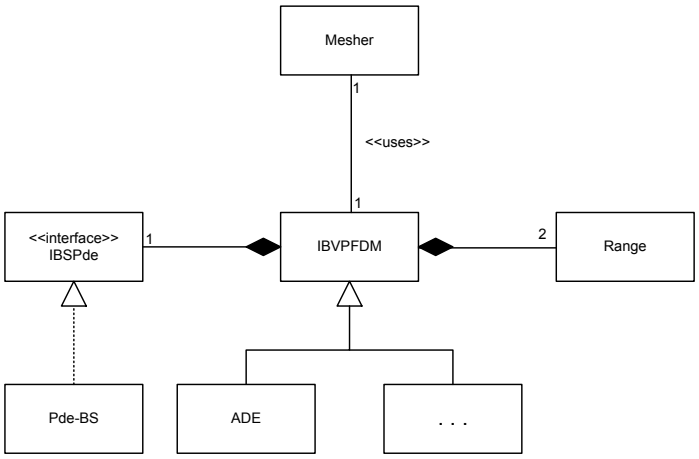

Figure 5 Extending the framework: ADE scheme

The assumption is that the model in Figure 2 can be adapted using *analogous reasoning* to use the software assets from Figure 5 to design a new PDE/FDM model. These are:

- The software architecture.
- The layering patterns that promote information hiding.
- The components and their interfaces.
- The data and control flow logic in the mediator components.
- The analogies between the mathematics used to model SDEs and PDEs.

Answer the following questions:

a) What is the *core process*, what is the system output /input and the steps that link them?
b) Find the major components and their responsibilities.
c) Determine the inter-component *provides/requires* interfaces.
d) Design PDE components using the *Layers* pattern.
e) Design FDM component using the *Layers* pattern and possibly the *Template Method pattern* (hint: find *variant* and *invariant behaviour* between the components that implement various finite difference schemes).
f) Determine a strategy for designing this system as a series of prototypes. Test each prototype.
g) Use the ideas relating to the use of interfaces and base classes when designing FDM hierarchies for SDEs to discover similar hierarchies for PDEs. Concentrate on interface specifications and code reusability. Consider where the *Template Method* could be used.