Chapter 15 Option Pricing Application using the Monte Carlo Method

Daniel J. Duffy dduffy@datasim.nl

See also

a) Same reusable design but now in C++

https://onlinelibrary.wiley.com/doi/abs/10.1002/wilm.10647

b) Background, relevant books

https://www.datasim.nl/books

c) Test your design knowledge by mapping Figure 15.1 to Python code, modules and packages. For parallel processing, use a Python actor model and Python multiprocessing ("manage processes like threads"). We can view the designs and code for the C# and C++ solutions as the requirements document for a Python solution. Low-risk and plain sailing. Use scipy *and* numpy *libraries for RNG, statistics, plotting etc.*

d) Four levels of reuse (taking C++ and C# as "sparring partners"):

   a. Build and document the C++ application.

   b. Use the design blueprints from a. to build the equivalent C# application.

   c. Port the C++ code from a. to generate the equivalent C# code.

   *d. Write a multi-language application using **interop libraries** that combine C++, C# and Python code. This is the **cross-language** application development.*

Draft 0.95 of forthcoming 2025 book by Daniel J. Duffy

Chapter 15 Option Pricing Application using the Monte Carlo Method

Why are software process models important? Primarily because they provide guidance on the order (phases, increments, prototypes, validation tasks) in which a project should carry out its major tasks. Many software projects have come to grief because they pursued their various development and evolution phases in the wrong order.

Barry Boehm

15.1 Introduction and Objectives

In this chapter we continue with the models from chapters 7 and 8 by applying them to a specific application, namely option pricing in computational finance using the Monte Carlo method. In this case we discuss how we implemented a solution using both C++ (two versions) and C#. The architectural groundwork and mathematical foundations for this problem has already been done and this fact allows us to focus on the detailed design patterns and their implementation in C#.

The main topics and remarks are:

1. Requirements are known (Duffy and Kienitz, 2009; Duffy, 2006; Duffy, 2018; Duffy, 2022). We have implemented Monte Carlo simulators in C++ over a period of several years. We modelled them as instance systems of the Resource Allocation and Tracking (RAT) category.
2. The C++ solution and design knowledge about the simulator will function as requirements input to the C# solution in this chapter. We do not discuss the mathematical aspects here.
3. We use several design patterns that have a major impact on the maintainability of the code, in particular *Mediator*, *Builder*, *Factory Method*, *Template Method* and delegate-based *Observer*.
4. The solution is a foundation upon which other applications can be built.

The main focus is to show how to design multiparadigm software systems. By *multiparadigm* we mean that we can choose the most appropriate course of action to take at each stage of the software lifecycle. In other words, we can decide which route to adopt based on certain criteria based on project resources and constraints. In order to promote flexibility and interoperability at all levels, we partition the design space into four conceptual attention areas or activities:

P1: Problem description and scope

Understand and scope the problem. Introduce suitable notation to define and describe the concepts and names in the problem. This is called *requirements analysis*.

This book is unique in the sense that we have *reusable domain models* that we can consult when designing a new software system (Duffy, 2004). This allows us to discover some of the system's major requirements.

P2: System Decomposition and Data Flow

We describe how the major input data is processed in a series of *activities* to produce the major system output which is what the customer wants to see. We partition the system into loosely-coupled and cohesive subsystems (modules) that we document as a *context diagram*.

P3: Coarse-grained logical Design

We refine and elaborate the modules from phase P2 by mapping them to language-independent system and architectural patterns (POSA 1996; Shaw and Garlan, 1996) that build flexibility into the design. These patterns that promote system maintainability, portability, efficiency and understandability. The product from this phase is set of detailed descriptions of evolving components that will eventually be the building blocks of the resulting software system. Noteworthy and strategic patterns in this phase are *Layers*, *Whole-Part*, *Mediator* and *Model-View-Controller*. This phase documents the evolving software as a set of *design blueprints*, for example UML *component diagrams* that we introduce in chapter 20.

P4: Fine-grained logical Design

In this phase we employ language-dependent patterns and idioms to help us design flexible class-based code. We are now in *design patterns* territory as discussed in the influential work GOF (1995). We broaden the scope and applicability of these traditional and somewhat outdated object-oriented design patterns by reimplementing them directly in modern C++, C# or Python or by porting GOF patterns to their functional and generic equivalents. We thus present the GOF patterns in their object-oriented, functional and generic forms and variants. Important design patterns that we use in this phase to promote the flexibility of software prototypes are *Factory Method*, *Builder*, *Composite*, *Strategy* and *Visitor*. This phase documents the evolving software as a set of design blueprints, for example UML *class diagrams* that we introduce in chapters 2, 9, 10 and 11.

P5: Physical Design and Choice of Programming Language

This is the phase in which we bring the software prototype into production. This can entail integrating the prototype with external systems such as hardware, networks and databases. A discussion of these topics is outside the scope of this book.

We also give an introduction to the *Actor* model and we show how it compares to OOP by writing a Monte Carlo simulator using the *Asynchronous Agents Library* in C++.

15.2 Background to Problem and Requirements

Monte Carlo methods are widely used in various fields of science, engineering, and mathematics, such as physics, chemistry, biology, statistics, artificial intelligence, finance, and cryptography. They have also been applied to social sciences, such as sociology, psychology, and political science. Monte Carlo methods have been recognised as one of the most important and influential ideas of the 20th century, and they have enabled many scientific and technological breakthroughs.

The basic algorithm that underlies Monte Carlo simulation is:

. Define a domain of possible inputs.

. Generate inputs randomly from a probability distribution over the domain.

. Perform a deterministic computation on the inputs.

. Aggregate the results.

The Monte Carlo is ubiquitous in many fields of science. In computational finance, for example it is used to compute quantities such as option prices and risk factors as discussed in Glasserman (2004), Duffy and Kienitz (2009) and Duffy (2018).

This chapter focuses on pricing one-factor options. Monte Carlo is useful because it always produces a solution even in cases when analytical solutions are not forthcoming.

15.3 Monte Carlo for the Impatient

Before we design a Monte Carlo (MC) simulator system to price (financial) options based on mathematical algorithms and domain architectures, we take a step backwards as it were by reducing the scope of the problem. We examine the *core process* and then we create a software prototype to test the no-frills algorithm that implements it. In this case the core process consists of the following activities:

1. Initialise market and model data.
2. Choose and create the *stochastic differential equation* (SDE) that models the random behaviour of stock prices. We discuss SDEs in section 15.4.1.
3. Use the data and objects from steps 1 and 2 as input to MC algorithm to compute option price.
4. Present the results from step 3 to external (decision-support) systems.
5. (Optional) management systems are informed of the progress of the computation in step 3 at regular intervals.

We present "get it working" prototype code in C++ in the following subsection. The most important requirement in this version of the software is that the algorithm produces accurate results. In this case we compare the analytical price of a plain option (using the Black-Scholes formula) with the value produced by the Monte Carlo simulator. For completeness, the code for the analytical price in Python is given by:

```python
# Initialise the option data
r = 0.08
d = 0.0
sig = 0.3
T = 0.25
K = 65.0
S_0 = 60


# Exact
import numpy as np
from scipy.stats import norm

N = norm.cdf

def CallPrice(S, K, T, r, sigma):
    d1 = (np.log(S/K) + (r + sigma**2/2)*T) / (sigma*np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    return S * N(d1) - K * np.exp(-r*T)* N(d2)

print ('Exact Call:  ', CallPrice(S_0, K, T, r, sig))

def PutPrice(S, K, T, r, sigma):
    d1 = (np.log(S/K) + (r + sigma**2/2)*T) / (sigma*np.sqrt(T))
    d2 = d1 - sigma* np.sqrt(T)
    return K*np.exp(-r*T)*N(-d2) - S*N(-d1)
print ('Exact Put:  ', PutPrice(S_0, K, T, r, sig))
```

It is possible to compute call and put prices of European for different values of the above parameters. This is useful because we need to test the accuracy of the Monte Carlo pricer for a range of values. Tests must be comprehensive. More generally, we can compare the price using the Monte Carlo simulator with finite difference (FDM) and lattice methods, for example. These methods are discussed in the books of Daniel J. Duffy, for example Duffy (2018) and Duffy (2022) where we discuss C++ implementations of finite difference methods as well as the actual underlying mathematical and numerical theory of FDM.

15.3.1 C++ Solution

Before we discuss the design of a maintainable Monte Carlo option pricer based on domain architectures we examine a minimalist version that implements the core process whose input is option data and that computes the price of a plain (European) option. The activities in the core process are 1) read option data, 2) simulate the stock paths and 3) compute option price. We map these activities to basic C++ code.

The basic option data is:

```
struct OptionData
{ // Option data needed for Black Scholes

        double K;
        double T;
        double r;
        double sig;

        // Extra data
        double D;               // dividend
        int type;               // 1 == call, -1 == put


        explicit constexpr OptionData(double strike, double expiration,
                double interestRate, double volatility, double dividend,
                                                        int PC)
                : K(strike), T(expiration), r(interestRate),
                        sig(volatility), D(dividend), type(PC) {}
};
```

The code that implements the above five steps is:

```
int main()
{
        std::cout <<  "1 factor MC with explicit Euler\n";

        // Step 1
        OptionData myOption{ 65.0, 0.25, 0.08, 0.3, 0.0, -1 };
        auto PayOffFunction = [&](auto S)
        { // Payoff function, Put

                  return std::max(myOption.K - S, 0.0);
                  // return std::max(S - myOption.K, 0.0);

        };

        // Step 2, see section 15.4.1 of book
        SDE sde(myOption);

        // Initial value of SDE
        double S_0 = 60;

        // Variables for underlying stock
        double VOld = S_0;
        double VNew;

        std::size_t NT = 100;
        std::cout << "Number of time steps: ";
        std::cin >> NT;

        // V2 mediator stuff
        long NSIM = 50000;
        std::cout << "Number of simulations: ";
        std::cin >> NSIM;
        double M = static_cast<double>(NSIM);
double dt = myOption.T / static_cast<double> (NT);
double sqrtdt = std::sqrt(dt);

// Normal random number
double dW;
double price = 0.0;    // Option price
double sumPriceT = 0.0;

// Normal (0,1) rng
std::default_random_engine dre;
std::normal_distribution<double> nor(0.0, 1.0);

StopWatch<> sw;
sw.Start();

// Calculate a path at each iteration of the MC draws
// Step 3
for (long i = 1; i <= NSIM; ++i)
{


        VOld = S_0;
        double t = 0.0;
        for (std::size_t index = 0; index < NT; ++index)
        {
                // Create a random number
                dW = nor(dre)*sqrtdt;
                VNew = VOld
                        + (dt * sde.drift(t, VOld))
                        +(sde.diffusion(t, VOld) * dW);
                VOld = VNew; t += dt;  }
                double payoffT = PayOffFunction(VNew);
                sumPriceT += payoffT;
        }

        // Finally, discounting the average price
        price = std::exp(-myOption.r * myOption.T) * sumPriceT/M;

        // Step 4
        std::cout << "Price, after discounting: "
                                      << price << ", " << std::endl;

        sw.Stop();
        // Step 5
        std::cout << "Elapsed time in seconds: " << sw.GetTime() << '\n';
        return 0;
}
```

15.4 RAT 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. We now 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 summarise 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 in 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 to the five steps in section 15.3:

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.

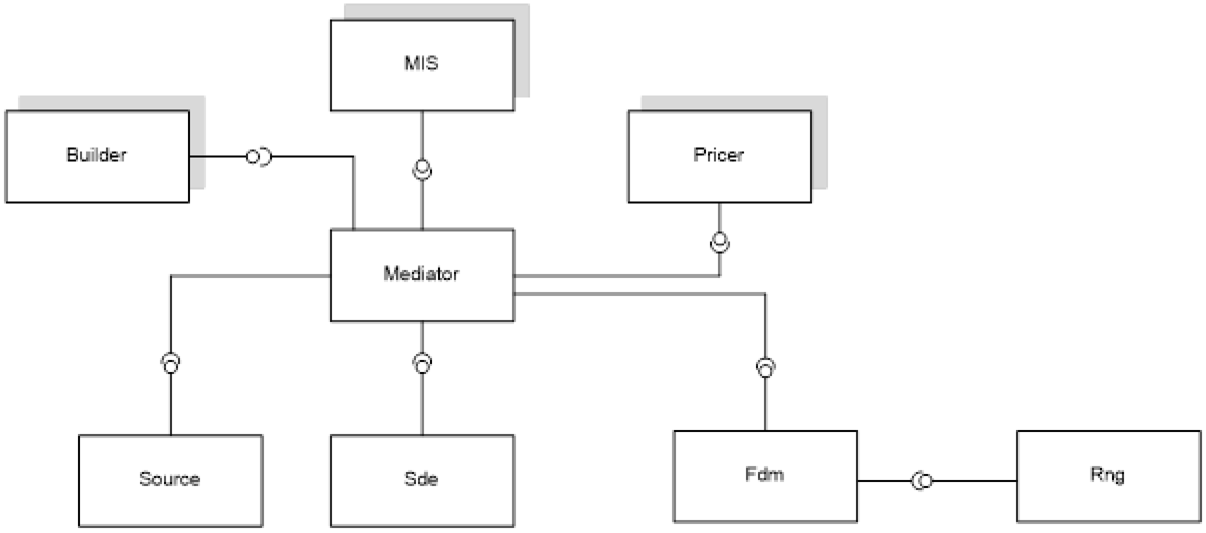

Figure 15.1 New Context diagram for Monte Carlo Engine

The context diagram for the current application is shown in Figure 15.1. 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 15.1:

. *Source*: The system containing the data relating to the request.

. *Sde*: The system that models stochastic differential equations. 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 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 (variates with mean zero and standard deviation one).

. *Pricer*: This system contains classes to price one-factor options using the Monte Carlo algorithm. 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 finally 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 that creates and initialises the modules and their structural relationships in Figure 15.1. The newly-created objects are encapsulated in .NET tuples.

. *Mediator*: This is the central coordinating entity that manages the data 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.

. *Rng*: a system to generate random numbers.

We now discuss how we implemented the modules in Figure 15.1 in C#. We deliberately avoid showing too much of the underlying mathematics as this is discussed elsewhere, for example Duffy and Kienitz (2009) and Duffy (2018).

15.4.1 The underlying Mathematics: Stochastic Differential Equations (SDEs)

In general, an SDE is uniquely specified if we define the drift, diffusion, Wiener increment and the initial condition of the stochastic process . 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 on which the sde is defined.
The initial value of the stochastic process .

In this chapter we model the above related group of characteristics in an interface. The interface contains abstract methods and properties:

```
using System;
using System.Collections.Generic;

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

    // Abstract methods, to be overridden
    double Drift(double x, double t);          // a (mu)
    double Diffusion(double x, double t);      // b (sig)

    // Some extra abstract methods associated with the SDE
    // Needed for FDM schemes (predictor-corrector, Milstein)
    double DriftCorrected(double x, double t, double B);
    double DiffusionDerivative(double x, double t);

    // Properties in the interface: X(0) and T (sde defined on (0,T)
    double InitialCondition { get; set; }
    double Expiration { get; set; }
}


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 driftCoeff, double diffusionCoeff,
                      double dividendYield, double initialCondition, double expiry)
    {
        mu = driftCoeff;
        vol = diffusionCoeff;
        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 Expiration
    {
        get
        {
            return exp;
        }
        set
        {
            exp = value;
        }
    }

}
```

15.4.2 Numerical Approximation of SDEs

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
    // IFdm is composed of an Sde instance
    ISde StochasticEquation
    {
        get;
        set;
    }

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

We see that clients of this interface and of the classes that implement the interface must provide a Wiener increment values as input to the `advance()` method. We now define a base class `FdmBase` 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
{ // Base FDM class containing data (e.g. mesh) that is common to all subclasses

    protected ISde sde;

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

    protected double dtSqrt;

    public FdmBase(ISde stochasticEquation, int numSubdivisions)
    {
        sde = stochasticEquation;
        NT = numSubdivisions;
        k = sde.Expiration / (double)NT;
        dtSqrt = Math.Sqrt(k);
        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;
        }

    }

    // Override the abstract method from IFdm
    public ISde StochasticEquation
    {
        get
        {
            return sde;
        }
        set
        {
            sde = value;
        }
    }

    // Advance from level n to level n+1
    public abstract double  advance(double  xn, double  tn, double  dt,
                                     double  WienerIncrement);
}
```

As a first example, the simplest finite difference scheme is probably the explicit Euler scheme. 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)
    {
         return xn + sde.Drift(xn, tn) * dt + sde.Diffusion(xn, tn) * dtSqrt * normalVar;
    }
}
```

Another popular scheme due to Milstein 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)
    {
         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);
    }
}
```

15.4.3 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.

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(((int)DateTime.Now.Ticks & 0x0000FFFF)); }

        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 || S <= 0.0);

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

and

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

       // Seed is from system clock
        public BoxMullerNet() { rand = new Random(((int)DateTime.Now.Ticks & 0x0000FFFF)); }
        double U1, U2;

        public override double GenerateRn()
        {
            // U1 and U2 should be independent uniform random numbers
            do
            {
                U1 = rand.NextDouble();   // In interval [0,1)
                U2 = rand.NextDouble();   // In interval [0,1)
            }
            while (U1 <= 0.0);

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

15.4.4 Option Pricers

We now describe the component that prices one-factor options using the Monte Carlo method. 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, a delegate
        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:

```
    public class EuropeanPricer : Pricer
    {
        private double price;
        private double 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.Write("Compute Plain price: ");
              price = DiscountFactor() * sum / NSim;
              Console.WriteLine("Price, #Sims: {0}, {1}",price,NSim);
        }

        public override double Price()
        {
            return price;
        }

    }
```

15.4.5 Putting it all together: The Mediator

After we designed the software components we then 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*. 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. The *Mediator* pattern is described in GOF (1995). The attention points are:

1. Create the components in Figure 15.1 that the mediator needs (these components are created by a builder).
2. Determine the *provided/required* 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 15.1. It has a constructor that accepts a tuple containing the components that it needs and it has public events that define the communication with the pricer and management components:

```
using System;
using System.Collections.Generic;

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

// Events to MIS system
public delegate void NotifyMIS(int n);

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

    // MIS notification
    private event NotifyMIS mis;

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

    // Event notification
    private event PathEvent<double> path;             // Signal to the Pricers
    private event EndOfSimulation<double> finish;     // Signals that data has been sent


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

        mis = i => { if ((i / 10000) * 10000 == i)
                                 Console.WriteLine("Iteration # {0}", i); };

        // Define slots for path information
        path = optionPaths;
        // Signal end of simulation
        finish = finishOptions;

        NSim = numberSimulations;
        res = new double[fdm.NT+1];

    }

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

        double VOld, VNew;

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

            // Give status after a given numbers of iterations
            mis(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());
                    res[n] = VNew; VOld = VNew;
                }

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

15.4.6 System Configuration and Interface Specification

Having designed the software modules we must now decide how to instantiate them. We instantiate the classes and then we add them to the end-product which is the network object in Figure 15.1. We generate random numbers using the Box-Muller method in combination with the .NET `Random` class. To this end, we instantiate the appropriate classes and configure the software system with these choices. More generally, we execute the following steps:

1. Initialise each of the modules and their data in Figure 15.1.
2. Connect the modules based on the *provided-required* 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 modules 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 store in an assembly which is loaded into memory at run-time.

In the interest of completeness, we should document the emergence of these new low-level classes by extending Figure 15.1 to form a more detailed design-level system context diagram. 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 to configure the most important modules in Figure 15.1. 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 this delegate type 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 15.1. It uses the `Console` to elicit input from the user. Furthermore, the builder uses and needs the parameters corresponding to well-known SDEs. `MCBuilder`' 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 `MCBuilder` 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
```

```
private double r;
private double v;
private double d;
private double T;
private double K;
private double IC;  // S_0

private event PathEvent<double> path;
private event EndOfSimulation<double> finish;

// Constructor (data important at this stage)
public MCBuilder(OptionData optionData)
{
    // r, div, sig, T, K, IC, NSim
    // 1   2    3   4  5   6   7    (Item*)
    r = optionData.r_;
    d = optionData.r_ - optionData.b_;
    v = optionData.sig_;
    T = optionData.T_;
    K = optionData.K_;
    IC = optionData.S_;
}

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);

     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);
    IPricer pricer = GetPricer(payoff, discounter);
    return new Tuple<ISde, FdmBase, IRng>(sde, fdm, rng);
}

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

    if (c == 1)
    { // GBM

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

        return new GBM(r, v, d, IC, T);
    }
}

private IRng GetRng()
{
    Console.WriteLine("Create RNG");
    Console.WriteLine("1. Box-Muller, 2. Polar Marsaglia ");

    int c = Convert.ToInt32(Console.ReadLine());

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

        case 2:
             rng = new PolarMarsagliaNet();
             break;

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

    return rng;
}

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

    FdmBase fdm;

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


    switch(c)
    {
        case 1:

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

        case 2:

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

        default:
            fdm = new EulerFdm(sde, NT);
            break;
    }

    return fdm;
}

private IPricer GetPricer(Payoff payoff,
                                         Func<double> discounter)
{
    // Choice here
    IPricer op = new EuropeanPricer(payoff, discounter);
    path += op.ProcessPath;
    finish += op.PostProcess;


    Payoff payoff2 = x => Math.Max(0.0, x - K);
    IPricer op2 = new EuropeanPricer(payoff2, discounter);
    path += op2.ProcessPath;
    finish += op2.PostProcess;

    return op;
}

public PathEvent<double> GetPaths()
{
    return path;
}
```

```
        public EndOfSimulation<double> GetEnd()
        {
            return finish;
        }

    }
```

15.4.7 Testing the Application

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.

. Choose a builder.

. Create the mediator.

. Create the payoff function.

. Create instances and make them observers of the mediator.

. Run the program and examine the output.

In the following, we create three sets of tests:

```
    // Main program etc.
    public class MCPricerApplication
    {


        public static void Main()
        {
            int NSim = 1000000;
            {

                Console.WriteLine("Batch 1"); // P = 5.845, C = 2.132
                // Get data from Source
                OptionData data =
                           new OptionData(0.08,0.3, 65.0, 60.0, 0.25, 0.08);
                data.print();

                MCBuilder<ISde, FdmBase, IRng> builder =
                           new MCBuilder<ISde, FdmBase, IRng>(data);
                var parts = builder.Parts();
                var path = builder.GetPaths();
                var finish = builder.GetEnd();
                MCMediator mcp =
                           new MCMediator(parts, path, finish, NSim);
                mcp.start();
            }

            {
                Console.WriteLine("Batch 2"); // P = 0.238, C = 9.755
                // Get data from Source
                OptionData data =
                                  new OptionData(0.2,0.1,100,100,0.5,0.2);
                data.print();

                MCBuilder<ISde, FdmBase, IRng> builder =
                                  new MCBuilder<ISde, FdmBase, IRng>(data);
                var parts = builder.Parts();
                var path = builder.GetPaths();
                var finish = builder.GetEnd();
                MCMediator mcp =
                                  new MCMediator(parts, path, finish, NSim);
                mcp.start();
            }
            {

                Console.WriteLine("Batch 3");
                // Get data from Source
                OptionData data =
                           new OptionData(0.06,0.3, 65.0, 60.0, 0.25, 0.08);
                data.print();

                MCBuilder<ISde, FdmBase, IRng> builder
                                  = new MCBuilder<ISde, FdmBase, IRng>(data);
                var parts = builder.Parts();
                var path = builder.GetPaths();
                var finish = builder.GetEnd();
                MCMediator mcp =
                                  new MCMediator(parts, path, finish, NSim);
                mcp.start();
            }

        }
    }
```

15.6 An Introduction to the Actor Model

An *actor* is a concurrent computational entity that can carry out the following actions in parallel:

. It can send a finite number of messages to other actors.

. It can create a finite number of new actors.

. It can designate the behaviour to be used for the next message that it receives.

There is no assumed sequence to the above actions and they can be carried out in parallel. A definition from Wikipedia is:

The actor model in computer science is a mathematical model of concurrent computation that treats an actor as the basic building block of concurrent computation. In response to a message it receives, an actor can: make local decisions, create more actors, send more messages, and determine how to respond to the next message received. Actors may modify their own private state, but can only affect each other indirectly through message (removing the need for lock-based synchronisation).

The actor model was inspired by physics and is has been influenced by programming languages such as *Lisp* and *Simula*. An actor is a computational agent with a mail address (mailbox name) and a behaviour. The mailbox determines the identity of an actor and it designates a buffer that can store an unbounded linear sequence of messages or *communications*. An actor's behaviour is determined by its actions in response to a communication. The only way that actors may affect each other is by communications. They respond to communications by sending messages, creating new actors and creating replacements.

Communications are contained in tasks. A *task* in this context is a 3-tuple consisting of the following elements:

. A tag or task ID that uniquely distinguishes the task from all other tasks in the system.

. A target, which is the mail address to which the communication is to be delivered.

. A communication containing information that is made available to the actor at the target when it gains possession of the given task.

We remark that a short introduction to the actor model was given in Duffy (1996). At the time, actor languages and frameworks were few and far between and the only alternative was an immature OOP technology. An actor language is a low-level language that can be used to create more complex and sophisticated languages. Actor languages support objects, abstraction and concurrency but not classes, inheritance or strong typing.

15.6.1 The *Asynchronous Agents Library*

The *Asynchronous Agents Library* (or *Agents Library* for short) provides both an actor-based programming model (see Agha, 1990) and message passing interfaces for coarse-grained dataflow and pipelining tasks. Asynchronous agents enable applications to make use of latency by performing work as other components wait for data.

An *asynchronous agent* (or *agent*) is an application component that communicates with other agents to solve larger computing tasks. The .NET *Concurrency Runtime* task scheduler provides an efficient mechanism to enable agents to block and yield cooperatively without requiring preemption.

An agent finds itself during its lifetime in various states:

. *created*: the agent has been scheduled for execution.

. *runnable*: the runtime is scheduling the agent for execution.

. *started*: the agent has started and is running.

. *done*: the agent has finished processing.

. *canceled*: the agent has been canceled before it entered the started state.

The *Asynchronous Agents Library* includes the following types of message blocks:

unbounded_buffer: a concurrent queue of unbounded size. In other words, it stores an unlimited number of messages.

overwrite_buffer: stores one message that can be written and read from multiple times. In other words, a single value can be updated many times.

single_assignment: stores one message that can be written to once and read from multiple times. In other words, a single value can be set just once.

call: a function that is invoked whenever a value is added to the message block.

transformer: a function that is invoked whenever a value is added to the message block; the function's return value is added to an output message block.

choice: selects a message from a set of sources.

join: waits for more than one source before proceeding.

multitype_join: same as join, except that inputs may have multiple message types.

timer: produces messages based on time intervals.

There is more functionality in this library but unfortunately a discussion is outside the current scope. We focus on a number of motivational examples.

15.6.2 Initial Examples

Sources and targets are two important participants in message passing. A *source* refers to an endpoint of communication that sends messages. A *target* refers to an endpoint of communication that receives messages. A source is an endpoint that we read from and a target as an endpoint that we write to. Applications connect sources and targets together to form *message networks*.

We give some initial examples of using the different message block types. We take the simple case of sending and receiving three characters that we conveniently encapsulate in a number of functions to reduce code duplication:

```
template <typename Buffer>
        void Send(Buffer& buffer)
{
        concurrency::send(buffer, 'a');
        concurrency::send(buffer, 'b');
        concurrency::send(buffer, 'c');
}

template <typename Buffer>
        void Receive(Buffer& buffer)
{
        std::cout << concurrency::receive(buffer) << ',';
        std::cout << concurrency::receive(buffer) << ',';
        std::cout << concurrency::receive(buffer) << '\n';
}

template <typename Buffer>
        void SendAndReceive(Buffer& buffer)
{
        Send(buffer);
        Receive(buffer);
}
```

We now show the use of three message block types:

```
// A message buffer that is shared by the agents.
concurrency::unbounded_buffer<char> buffer1;
concurrency::overwrite_buffer<char> buffer2;
```

```
        concurrency::single_assignment<char> buffer3;

        SendAndReceive<concurrency::unbounded_buffer<char>>(buffer1);
        // a,b,c

        SendAndReceive<concurrency::overwrite_buffer<char>>(buffer2);
        // c,c,c

        SendAndReceive<concurrency::single_assignment<char>>(buffer3);
        // a,a,a
```

The next example shows how to create a callback and call it when an event fires:

```
        // 'call' class. Perform a work function when data is received
        // 'event' is a synchronisation object for flow of execution
        concurrency::event receivedAll;

        int receiveCount = 0; int maxReceiveCount = 3;
        concurrency::call<char> callback = [&](char c)
        {
                std::cout << c << ",";
                if (++receiveCount == maxReceiveCount)
                {
                        receivedAll.set(); // set event to signalled state
                }
        };

        Send(callback);
        // Wait for call object to process all items
        receivedAll.wait();     // a,b,c
```

The next example how to transform input to output (and they can be of different types):

```
        // Transformer on different input and output types
        std::cout << '\n';
        concurrency::transformer<char,int> converter = [](char c)
        {
                return int(c); // simple test
        };

        SendAndReceive(converter);
```

We conclude this section by introducing the `choice` and `join` classes. Both are concerned with the processing of a set of messages. In the first case we select among competing tasks:

```
        // Choice: select the first available messages from a set of sources
        concurrency::overwrite_buffer<long> rel1;
        concurrency::overwrite_buffer<long> rel2;
        concurrency::single_assignment<long> rel3;

        auto selectOne = concurrency::make_choice(&rel1, &rel2, &rel3);

        concurrency::parallel_invoke(
                [&rel1] {concurrency::send(rel1, relation(22)); },
                [&rel2] {concurrency::send(rel2, relation(14)); },
                [&rel3] {concurrency::send(rel3, relation(13)); }
        );

        // See who responds first
        switch (concurrency::receive(selectOne))
        {
                case 0:
                        std::cout << "rel1 " << concurrency::receive(rel1) << '\n';
                        break;
                case 1:
                        std::cout << "rel2 " << concurrency::receive(rel2) << '\n';
                        break;
                case 2:
                        std::cout << "rel3 " << concurrency::receive(rel3) << '\n';
                        break;
                default:
                        std::cout << "oops\n";
                        break;
        }
```

In the second case run several agents in parallel and we wait on them before proceeding:

```
        // Run multipe tasks
        concurrency::single_assignment<long> relA;
        concurrency::single_assignment<long> relB;
        concurrency::single_assignment<long> relC;

        auto joinAll = concurrency::make_join(&relA, &relB, &relC);
        concurrency::parallel_invoke(
                [&relA] {concurrency::send(relA, relation(6)); },
                [&relB] {concurrency::send(relB, relation(7)); },
                [&relC] {concurrency::send(relC, relation(10)); }
        );

        auto result = concurrency::receive(joinAll);

        std::cout << "relA " << std::get<0>(result)
                        << ", relB " << std::get<1>(result)
                        << ", relC " << std::get<2>(result) << '\n';
```

The output from the complete code in this section is:

```
        a,b,c
        c,c,c
        a,a,a
        a,b,c,
        97,98,99
        rel3 233
        relA 8, relB 13, relC 55
```

15.6.3 *Producer-Consumer* Pattern and Work Flow

The *Agents* library can be used to implement the *producer-consumer* pattern more easily than is possible using threads and synchronising queues (see Demming and Duffy, 2010). In the current case we do not have to worry about locking of shared data because there is no shared data and agents communicate via message passing. In general, the producer sends messages to a message block while the consumer reads from that block. Two common scenarios are: 1) the consumer must receive each message that the producer sends, and 2) the consumer periodically polls for data and hence it does not have to receive each message.

The example that we take here simulates a stock quotes application. Data is read in array batches from an external device and each item in the array is sent to a consumer to be displayed in Excel, console or other output device. Both agent classes are derived from the abstract base class `concurrency::agent` with pure virtual member function `run()`. We use `ITarget` and `ISource` to denote the communication endpoints for producer and consumer, respectively. A producer sends data to a target and a consumer receives data from a source.

The producer class is:

```
class Producer : public concurrency::agent
{
private:
        // The target buffer to write to.
        concurrency::ITarget<double>& _target;
        std::vector<double> _quotes;
public:
        explicit Producer(concurrency::ITarget<double>& target,
                              std::vector<double>& quotes)
                : _target(target), _quotes(quotes)
        {
                std::cout << "Agent state in ctor (created, not scheduled): "
                              << this->status() << '\n'; // 0
        }

        void run()
        {
                std::cout << "Agent state (started and running): "
                              << this->status() << '\n'; // 2
                // For illustration, create a predefined array of stock quotes.
                // A real-world application would read these from an external
                // source, such as a network connection or a database.

                // Send each quote to the target buffer.
                std::for_each(begin(_quotes), end(_quotes), [&](double quote)
                {

                        send(_target, quote);

                        // Pause before sending the next quote.
                        concurrency::wait(20);
                });
                // Send a negative value to indicate the end of processing.
                concurrency::send(_target, -1.0);

                // Set the agent to the finished state.
                done();
        }

};
```

The consumer class is:

```
class Consumer : public concurrency::agent
{
private:
        // The source buffer to read from.
        concurrency::ISource<double>& _source;
public:
        explicit Consumer(concurrency::ISource<double>& source)
                : _source(source){}

        void run()
        {
                double sum = 0.0;
                std::cout << "\n New Consumer " << '\n';
                // Read quotes from the source buffer until we receive
                // a negative value.
                double quote;

                // Poll to see if any new values have arrrived
                while ((quote = concurrency::receive(_source)) >= 0.0)
                {
                        // Print the quote.
                        std::cout << "Current quote is " << quote  << '\n';
                        sum += quote;

                        // Pause before reading the next quote.
                        concurrency::wait(10);
                }
                std::cout << "Sum is " << sum << '\n'; // 21

                // Set the agent to the finished state.
                done();
        }

};
```

Finally, the test program is:

```
int main()
{

        // Data to be transferred from producer to consumer
        std::vector<double> quotes = { 1,2,3,4,5,6};

        // A message buffer that is shared by the agents.
        // Contains a single value that can be updated many times
        // message not removed from buffer
        //concurrency::overwrite_buffer<double> buffer;

        // message added to and removed from a shared buffer
        concurrency::unbounded_buffer<double> buffer;
        Producer producer(buffer, quotes);
        std::cout << "Agent state in main() (created, not scheduled): "
                        << producer.status() << '\n'; // 0
        Consumer consumer(buffer);

        // Start the agents and make them running
        producer.start();
        consumer.start();

        std::cout << "Agent state(runnable): "<<producer.status() << '\n'; // 1
        // Wait for the agents to finish.
        concurrency::agent::wait(&producer);
        std::cout << "Agent state (done): " << producer.status() << '\n'; // 3
        concurrency::agent::wait(&consumer);
}
```

This simple design contains many of the features that we can use when writing code based on the actor model. An agent is implemented as a C++ class and it is composed of one or more targets and sources (in this case `concurrency::ITarget<double>` and `concurrency::ISource<double>`). An actor is an isolated unit of state and logic and it embodies a stronger form of information hiding. Each actor must implement the `run()` method (which is called by `concurrency::agent::start()`) where all the action takes place, in particular this is where sources and targets exchange data. An agent signals that it has finished processing by calling `done()`.

15.6.4 Concluding Remarks

We decided to introduce the *Agents* library as it fills a gap in the design panorama. In this case we are interested in event-based distributed systems. The library can be used in applications that use the traditional *Observer* pattern or the Boost *signals2* library. The major difference is that we do not call object methods but that agents send and receive data using messages.

We conclude this section with a list of features when programing with agents and tips to improve efficiency:

. Asynchronous agents are most effective when they isolate their internal state from other agents. This helps reduce contention on shared memory. State isolation reduces the chance of deadlock and race conditions because components do not have to synchronise access to shared data.

. Some buffer types (for example, `concurrency::unbounded_buffer<char>)` can hold an unlimited number of messages. A consequence is that an application can enter a low-memory or out-of-memory state if the producer sends messages to a data pipeline faster than the consumer can process them. We should then limit the number of active messages by the introduction of a *throttling mechanism* (for example, a *semaphore*) to limit the number of messages that are concurrently active in the data pipeline.

. The work in a data pipeline should be *coarse-grained*. This is because message-passing involves more overhead than fine-grained work that task groups are more suitable for. We can combine the two approaches by defining a coarse-grained data network that uses fine-grained parallelism at each processing stage. In short, we should not define small pipeline stages.

. Avoid passing large message payloads by value. For example, `overwrite_buffer<char>` offers a copy of every message that it receives to each of its targets. You can use pointers or references to improve memory performance when we pass messages that have a large payload.

. In some cases it is not clear when or where to deallocate memory as messages travel in a data pipeline. In particular, we should use `std::shared_ptr` when ownership is undefined.

15.7 Monte Carlo Simulation using the Actor Model

We have given an overview of the salient features of the *Agents* library and we have given some examples of how to use it. We now attempt to write a simple application using the features in the library in order to port the C++ code for this problem to C++ agents. As we see, we also implemented the Monte Carlo simulator in C# based on the RAT domain category. It is advantageous to adopt a modular approach and we implement each module as an actor. The challenge is to define a mapping from C++ objects to actors. To this end, we design the problem as a *producer-consumer* design pattern (Mattson, Sanders and Massingill, (2005); Demming and Duffy, (2010); Campbell and Miller, (2001)). In this case, the producer creates simulated stock data which is then consumer by a module that computes an option price, for example. We also include an MIS agent that informs us on how the algorithm is progressing. We examine the following modules (agents):

. `PathEvolver`: basic agent that produces values.
. `OptionPricer`: basic agent that consumes values.
. `MIS`: agent that gathers statistics on running program.

We model the producer class based on the C++ and C# code that we have already created in this chapter as well as on the motivating code in section 15.7.1. This agent is composed of three targets that are instantiated in client code (for example, `main()`) as we shall see:

```
// Demonstrates a basic agent that produces values.
class PathEvolver : public concurrency::agent
{
private:
        // The target buffer to write to.
        concurrency::ITarget<double>& _target;
        // The sentinel value
        concurrency::ITarget<bool>& _stop;

        // MIS
        concurrency::ITarget<std::size_t>& _mis;

        int NT;
        int NSIM;

        std::shared_ptr<Sde> sde;
        double S0;


public:
        explicit PathEvolver(concurrency::ITarget<double>& target,
                                        concurrency::ITarget<bool>& stop,

        concurrency::ITarget<std::size_t>& misStatus,int NSteps,
                int NSimulations, std::shared_ptr<Sde>& stochasticDE,       double S_0)
                : _target(target), _stop(stop), _mis(misStatus), NT(NSteps),
                                NSIM(NSimulations), sde(stochasticDE), S0(S_0) {}


        void run()
        {
                ThreadSafePrint("Starting producer");
                concurrency::send(_stop, false);
                std::default_random_engine dre;
                std::random_device rd;
                dre.seed(rd());
                std::normal_distribution<double> nor(0.0, 1.0);

                double k = sde->expiration() / static_cast<double>(NT);
                double sqrk = std::sqrt(k);
                double v;

                for (std::size_t n = 0; n < NSIM; ++n)
                {
                        concurrency::send(_mis, n);

                        double VOld = S0; double VNew;
                        double x = 0.0;

                        for (long index = 0; index < NT; ++index)
                        {
                                v = nor(dre);
                                // The FDM (in this case explicit Euler)
                                VNew = VOld + (k * sde->drift(x, VOld))
                                                + (sqrk * sde->diffusion(x, VOld) * v);

                                VOld = VNew;
                                x += k;

                        }
                        concurrency::send(_target, VNew);
                }
                concurrency::send(_stop, true);

                // Set the agent to the finished state.
```

```
               done();
               ThreadSafePrint("Exit producer");

};
```

The consumer agent receives data from the producer for as long as the producer has data to send.

```
// Demonstrates a basic agent that consumes values.
class OptionPricer : public concurrency::agent
{
private:
       // The source buffer to read from.
       concurrency::ISource<double>& _source;

       // Send sentinel value to target
       concurrency::ISource<bool>& _stop;

       // ...

       OptionData myOption;
       double price;

public:
       explicit OptionPricer(concurrency::ISource<double>& source, `
                      concurrency::ISource<bool>& stop, OptionData opt)
               : _source(source), _stop(stop), myOption(opt),price(0.0) {}

       double Price()
       { // Option price

               return price;
       }

       void run()
       {

               double payoffT;
               double sumPriceT = 0.0;
               int NSIM = 0;
               double VNew;

               bool stop;
               while ((stop = concurrency::receive(_stop)) == false)
               {
                      // Assemble quantities (postprocessing)
                      VNew = concurrency::receive(_source);
                      payoffT = myOption.myPayOffFunction(VNew);
                      sumPriceT += payoffT;

                      NSIM++;
               }


               // Finally, discounting the average price
               price = std::exp(-myOption.r * myOption.T) *
                                              sumPriceT / static_cast<double>(NSIM);

               done();
               ThreadSafePrint("Exit consumer");
       }

};
```

Finally, we define a management agent that monitors progress:

```
// agent that gathers statistics on running program
class MIS : public concurrency::agent
{
private:
       concurrency::ISource<std::size_t>& _mis;
```

```
        // Send sentinel value to target
        concurrency::ISource<bool>& _stop;

        std::size_t freq;

public:
        explicit MIS(concurrency::ISource<std::size_t>& mis,
                        concurrency::ISource<bool>& stop, std::size_t frequency)
                                : _mis(mis), _stop(stop), freq(frequency) {}

        void run()
        {
                bool stop;
                std::size_t counter;
                while ((stop = concurrency::receive(_stop)) == false)
                {
                        counter = concurrency::receive(_mis);
                        if ((counter / freq) * freq == ++counter)
                        {
                                std::cout << counter << ",";
                        }
                }

                done();
                ThreadSafePrint("Exit MIS");
        }
};
```

15.7.1 A Test Case

We now show how to create the agents and run the program. We use the Boost C++ *Parameter* library that allows us to define data using named variables:

```
        #include <boost/parameter.hpp>

        namespace OptionParams
        {
                BOOST_PARAMETER_KEYWORD(Tag, strike)
                BOOST_PARAMETER_KEYWORD(Tag, expiration)
                BOOST_PARAMETER_KEYWORD(Tag, interestRate)
                BOOST_PARAMETER_KEYWORD(Tag, volatility)
                BOOST_PARAMETER_KEYWORD(Tag, dividend)
                BOOST_PARAMETER_KEYWORD(Tag, optionType)
        }


        // Encapsulate all data in one place
        struct OptionData
        { // Option data + behaviour

                double K;
                double T;
                double r;
                double sig;

                // Extra data
                double D;               // dividend

                int type;               // 1 == call, -1 == put

                explicit constexpr OptionData(double strike,
                                double expiration, double interestRate,
                                double volatility, double dividend, int PC)
                        : K(strike), T(expiration), r(interestRate), sig(volatility),
                                              D(dividend), type(PC) {}

                template <typename ArgPack> OptionData(const ArgPack& args)
                {
                        K = args[OptionParams::strike];
                        T = args[OptionParams::expiration];
                        r = args[OptionParams::interestRate];
                        sig = args[OptionParams::volatility];
                        D = args[OptionParams::dividend];
                        type = args[OptionParams::optionType];

                        std::cout << "K " << K << ", T " << T << ",r " << r << std::endl;
                        std::cout << "vol " <<sig<< ", div " <<D<< ",type "<< type << std::endl;
                }

                double myPayOffFunction(double S)
                { // Payoff function

                        if (type == 1)
                        { // Call

                                return std::max(S - K, 0.0);
                        }
                        else
                        { // Put

                                return std::max (K - S, 0.0);
                        }
                }
        };
```

The following code involves one producer and three consumers which all use the data except the strike price. This allows us to define a single buffer from which these consumers receive data:

```
int main()
{

        // A message buffer that is shared by the agents.
        concurrency::overwrite_buffer<double> buffer;
        concurrency::overwrite_buffer<bool> sentinel;
        // Communication with MIS agent (draw count)
        concurrency::overwrite_buffer<std::size_t> misCount;

        // Create and start the producer and consumer agents.
        OptionData myOption1((
                OptionParams::strike = 50.0, OptionParams::expiration = 0.25,
                OptionParams::volatility = 0.3, OptionParams::dividend = 0.0,
                OptionParams::optionType = -1, OptionParams::interestRate=0.08));

        OptionData myOption2((
                OptionParams::strike = 55.0, OptionParams::expiration = 0.25,
                OptionParams::volatility = 0.3, OptionParams::dividend = 0.0,
                OptionParams::optionType = -1, OptionParams::interestRate=0.08));

        OptionData myOption3((
                OptionParams::strike = 65.0, OptionParams::expiration = 0.25,
                OptionParams::volatility = 0.3, OptionParams::dividend = 0.0,
                OptionParams::optionType = -1, OptionParams::interestRate=0.08));


        // Modify to suit your needs
        int NT = 500;
        int NSIM = 1'000'000;

        StopWatch<> sw;
        sw.Start();

        // Create SDE
        auto sde1 = std::shared_ptr<Sde>(new Sde(myOption1));
        double S0 = 60.0;
        PathEvolver producer(buffer, sentinel, misCount, NT, NSIM, sde1, S0);

        // Define pricers
        OptionPricer consumer1(buffer, sentinel,myOption1);
        OptionPricer consumer2(buffer, sentinel, myOption2);
        OptionPricer consumer3(buffer, sentinel, myOption3);

        int frequency = 100'000;
        MIS mis(misCount, sentinel, frequency);

        producer.start();
        consumer1.start();
        consumer2.start();
        consumer3.start();

        mis.start();

        // Wait for the agents to finish.
        concurrency::agent::wait(&producer);
        concurrency::agent::wait(&consumer1);
        concurrency::agent::wait(&consumer2);
        concurrency::agent::wait(&consumer3);
        concurrency::agent::wait(&mis);

        sw.Stop();
        std::cout << "Elapsed time, Raw creation: " << sw.GetTime() << '\n';

        std::cout << "Price: "  << '\n' << consumer1.Price() << '\n';
        std::cout << "Price: " << '\n' << consumer2.Price() << '\n';
        std::cout << "Price: " << '\n' << consumer3.Price() << '\n';
}
```

15.8 Summary and Conclusions

In this chapter we designed and implemented a complete application in C#. It is a compute-intense instance of the RAT (*Resource Allocation and Tracking*) domain category. In this case the application computes one-factor option prices using the Monte Carlo method. In fact, we originally designed the application in C++ as a RAT instance and then we ported it to C#. This is invaluable experience because we begin to see patterns in our designs, making it easier to design new applications that we have not seen before. In fact, we can *morph* the Monte Carlo application as it were into the new application while preserving the original intent of the context diagram Figure 15.1.

We also introduced the *Actor* model of computation that is concerned with data-flow and producer-consumer architectures. We gave some examples of use and we showed how to implement Monte Carlo method using actors. This gives us a new perspective on software design which gives us a viable alternative to the *call-and-return* architecture that is used in OOP. Another alternative is the component model that we introduce in chapter 20.

The Monte Carlo application that we design in this chapter can be seen as a prototype for similar applications in the same category, for example. We can use this knowledge when embarking on new software projects.

15.9 Exercises, Quizzes and Projects

1. Design Patterns that are motivated by Cognitive Linguistics

In sections 15.4.1, 15.4.2 and 15.4.3 we created three object hierarchies for stochastic differential equations, finite difference methods and random number generators. We designed them based on a three-level strategy that has a similarities with category judgements with hierarchies (Eysenck and

Keane, 2000; Rosch, Mervis, Gray and Boyes-Braem (1976)). A *category* is a number of objects which are considered equivalent. They can be designated by names, for example *clothing*, *furniture* and *vehicle*. A *taxonomy* is a system by which categories are related to one another by class inclusion. Each category within a taxonomy is entirely included within one other category. In other words, the *level of abstraction* within a taxonomy refers to a particular level of inclusiveness.

Thus, categories in a taxonomy are related to one another via class inclusion, with the highest level of abstraction being the most inclusive and the lowest level of abstraction being the least inclusive:

. *Superordinate level* (genus): the highest and most inclusive level of abstraction (generality). General designations for very general categories, such as Furniture and Flower. Informativeness is lacking at this level because few attributes are conveyed.

. *Basic level* (species): the middle level of abstraction. This seems to be the most cognitively efficient level. Basic level categories have high within-category similarities and high between-category dissimilarities. Category exemplars share a generalised identifiable shape. Adults regularly use basic level object names and children learn these names first. Examples are Chair and Rose.

. *Subordinate level* (specific objects): the lowest level of abstraction. This level exhibits the highest degree of specificity and within-category similarity. Economy is lacking at this level because many attributes are conveyed. Examples are my favourite Armchair and DogRose.

The class hierarchies in sections 15.4.1, 15.4.2 and 15.4.3 might suggest similarities with the above levels. Some initials impressions are:

. The superordinate level is realised by *interfaces*. An interface has no data and no executable behaviour (its methods are abstract).

. The basic level corresponds to *abstract classes*. An abstract class may have data (attributes) and a combination of abstract and non-abstract methods.

. The subordinate level corresponds to *concrete classes* that can be instantiated. They override and implement the abstract methods from the basic level.

The objective of exercises 1 and 2 is to combine the above knowledge concerning conceptual hierarchies and their equivalent realisations in sections 15.4.1, 15.4.2 and 15.4.3 to help us design semantically correct classes and class hierarchies.

Answer the following questions:

a) The class hierarchies in sections 15.4.2 and 15.4.3 can be directly mapped to the above conceptual hierarchies. Discuss the accuracy of this statement.
b) The class hierarchy in sections 15.4.1 does not contain classes corresponding to the Basic level. What are the consequences for software reusability? For example, what will you do when classes at the subordinate level share common data? Would it not have been more clever we had created classes at the Base classes in the first place? We are thinking specifically about the method `DriftCorrected()` which is essentially an example of the *Template Method* pattern.

c) Categorise the following concepts according to the above three-level system: Shape, Circle, Line, Polyline, Rectangle, Polygon, Closed Shape, Point, Ellipse and Triangle. Determine the criteria for a concept to be at a given level. Can some of these concepts be subsumed into more general ones?
d) Consider the concepts Polyline, Polygon, Rectangle and Square. Under which circumstances can they be implemented special cases of a single class? What are the advantages and disadvantages of this approach?
e) Categorise the following concepts and classes that we discussed in chapter 14 according to the above three-level system: Room, Shed, House, Hallway, Window, Kitchen, Building.

2. Conceptual Categories, Design and Design Patterns and Implementation.

The following exercises are meant to stimulate questions and discussion on a range of topics. The goal is to anything to help you design semantically correct and maintainable software.

Answer the following questions:

a) The GOF patterns in C++ use *delegation* so that a client `C` can call a method in another base class `B`. In many cases `B` is an abstract class containing a combination of abstract and non-abstract methods as well as member data. This introduces unnecessary coupling and for this reason we attempt to produce a C++ design based on the C# designs in section 15.4.2. This is a three-layer model and the major challenge is to model the following C# interface using C++ *Concepts*, that is:

```
public interface IFdm
{
    ISde StochasticEquation
    { get; set; }

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

Create the corresponding C++ concept for this interface. It will need to be a template class.

b) Apply the same techniques from part a) to a well-known GOF design pattern of your choice, for example *Strategy*.
c) Develop factory patterns for the C++ code corresponding to the C# code in sections 15.4.1, 15.4.2 and 15.4.3.