18 Software Architectures and Patterns for Pricing Applications

18.1 Introduction and Objectives

In this chapter we give an introduction to software design based on system decomposition techniques (Duffy 2004) and design patterns (GOF 1995). We focus on those patterns that have proven to be most useful in computational finance and that are used to create flexible, reliable and maintainable fixed income applications. Due to the complexity of such applications and the dependencies in these kinds of software systems we have found it necessary to design applications as networks of cohesive, loosely-coupled components and software modules. This approach subsumes *functional decomposition* techniques and the design patterns approach to software development.
The goal of this chapter is to show how apply design patterns to the applications that we have discussed in the first 17 chapters of this book. We discuss how flexible the proposed solutions are. In particular, we examine the changes that need to be made in the proposed software solutions in order to satisfy typical requirements. To this end, we give a quick review of design patterns. We then give an overview of the major high-priority design patterns that we use in fixed income applications. We first describe the traditional GOF patterns and we also discuss some of their limitations. We resolve these limitations by introducing the .NET *delegates* mechanism. This allows us to implement quite a few patterns in GOF 1995 using delegates and their use leads to loosely-coupled software systems.
We recall that we have GOF *Visitor* and *Strategy* patterns in chapters 4 and 9, respectively.
This chapter focusus mainly on the object-oriented programming model.

18.2 An Overview of the GOF Pattern

The origins of design patterns for software systems date back the 1980's and 1970's. It was not until 1995 that they were published by Eric Gamma and co-authors (GOF 1995). This influential book spurred interest in the application of design patterns to software development projects in C++ and Smalltalk.
The motivation for using design patterns originated from the work of architect Christopher Alexander:

"*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a millions times over, without ever doing it the same way twice.*"

The current authors have been working with design patterns since 1993 and we have applied them in different kinds of applications such as Computer Aided Design (CAD) and computer graphics, process control, real time and finance applications. Once you learn how a pattern works in a certain context, you will find that it easy to apply in new situations. The GOF patterns are applicable to objects and to this end they model the *object lifecycle*, namely object creation, the structuring of objects into larger configurations and finally modelling how objects communicate with each other using *message passing*. The main categories are:

- *Creational*: these patterns abstract the instantiation (object creation) process. The added-value of these patterns is that they ensure that an application can use objects without having to be concerned with how these objects are created, composed or internally represented. To this end, we create dedicated classes whose instances (objects) have the sole responsibility for creating other objects. In other words, instead of creating all our objects in `main()` (for example) we can delegate the object creation process to dedicated *factory objects*. This approach promotes the *single responsability principle*.
  The specific creational patterns are:
  - *Builder* (for complex objects that we create in a step-by-step manner).
  - *Factory Method* (define an interface for creating an object).
  - *Abstract Factory* (defines an interface for creating hierarchies of objects or families of related objects).
  - *Prototype* (create an object as a copy of some other object).
  - *Singleton* (create a class that has only one instance).

- *Structural*: these patterns compose classes and objects to form larger structures. We realise these class relationships by the appropriate application of structural modelling techniques such as *inheritance*, *association*, *aggregation* and *composition*.
  The structural patterns are:
  - *Composite* (recursive aggregates and tree structures).
  - *Adapter* (convert the interface of a class into another interface that clients expect).
  - *Facade* (define a unified interface to a system instead of having to access the objects in the system directly).
  - *Bridge* (a class that has multiple implementations).
  - *Decorator* (add additional responsibilities to an object at run-time).
  - *Flyweight* (an object that is shared among other objects).
  - *Proxy* (an object that is a surrogate/placeholder for another object to control access to it).
- *Behavioural*: these are patterns that are concerned with inter-object communication, in particular the implementation of algorithms and the sharing of responsibilities between objects. These patterns describe run-time control and data flow in an application. We can further partition these patterns as follows:
  - *Variations*: patterns that customise the member functions of a class in some way. In general, these patterns externalise the code that implements member functions. The main patterns are:
    * *Strategy* (families of interchangeable algorithms).
    * *Template Method* (define the skeleton of an algorithm in a base class; some variant steps are delegated to derived classes; common functionality is defined in the base class).
    * *Command* (encapsulate a request as an object; execute the command).
    * *State* (allows an object to change behaviour when its internal state changes).
    * *Iterator* (provide a means to access the elements of an aggregate object in a sequential way without
      exposing its internal representation).
  - *Notifications*: these patterns define and maintain dependencies between objects:
    * *Observer* (define one-to-many dependency between a *publisher* object and its dependent *subscribers*).
    * *Mediator* (define an object that allows objects to communicate without being aware of each other; this pattern promotes *loose coupling*).
    * *Chain of Responsibility* (avoid coupling between s*ender* and *receiver* objects when sending requests; give more than one object a chance to handle the request).
  - *Extensions*: patterns that allow us to add new functionality (in the form of member functions) to classes in a class hierarchy. There is only one such pattern:
    * *Visitor* (define an operation on the classes in a class hierarchy in a non-intrusive way).
    There are some other, somewhat less important behavioural patterns in GOF 1995:
- *Memento* (capture and externalise an object's internal state so that it can be restored later).
- *Interpreter* (Given a language, define a representation for its grammar and define an interpreter to interpret sentences in the language).

Which GOF patterns are useful when developing applications? An initial answer is that 20% of the patterns are responsible for 80% of developer productivity in our experience. We describe some of the most important patterns in the rest of this chapter.

18.3 Creational Patterns

A creational pattern is realised by a class (or a hierarchy of classes) whose instances are responsible for the creation of other objects. The former objects are specialised *factories* whose main responsibility is to create objects that will subsequently be used by client code.
There are a number of concerns when discovering the most appropriate patterns to use in an application, assuming of course that that the context requires it:

- Separating the object construction process from the clients that use objects.

- The object lifecycle policy.

We need to identify the reasons why we wish to use a given creational pattern. In this book we use the *Builder* pattern that creates complex aggregate objects and object networks and the *Factory Method* pattern that offers an interface to create instances of specific classes.

18.4 Builder Pattern

The *Builder* pattern is in a league of its own as it were because - in contrast to other creational patterns - it is used for the creation of complex objects and for the configuration of objects in a complete application. By 'complex' we mean any of the following:

- Whole-part hierarchies.
- The agent in a *PAC* pattern (see POSA 1996).
- The components of a *PAC* agent.
- Composites and recursive aggregates.
- The complete application (a network of objects).

The last example pertains to creating and initialising all the objects in the application. The *Builder* pattern offers many advantages:

- It takes care of the tedious and potentially unsafe work of creating data, objects and links between objects. Clients do not have to know how the objects are created and how the links are realised. In GOF terminology, it is stated as:

  'Builder separates the construction of a complex object from its representation so that the same construction process can create different representations'

- The *Builder* pattern is particularly useful when we create a Monte Carlo application. The `main()` function will delegate to a builder object, thus making the code easier to maintain and to understand.

This strategic pattern is useful for configuring arbitrary object networks, whole-part assemblies and composites. It is documented in GOF 1995 and we see it as a special case (or instance system) of the *manufacturing domain architecture* (code name MAN) that we introduced in Duffy 2004. In all cases we are interested in creating a product based on some given input data. We sometimes speak of *raw materials* when referring to the input data. The best way to understand the *Builder* pattern is to consider it to be the implementation of a process that creates products from raw materials. It is important to focus on the *data flow* aspects of the process. To this end, the process is broken down into three major activities:

- *Processing*: parses raw materials and creates the building blocks that will form the final product. This phase is implemented by a *Director* object.
- *Conversion*: creates the final products by assembling its parts. This phase is implemented by a *Builder* object. This is a step-by step process.
- *Postprocessing*: optimises the products and formats it so that it can be used by client systems. This system produces the *final Product*.

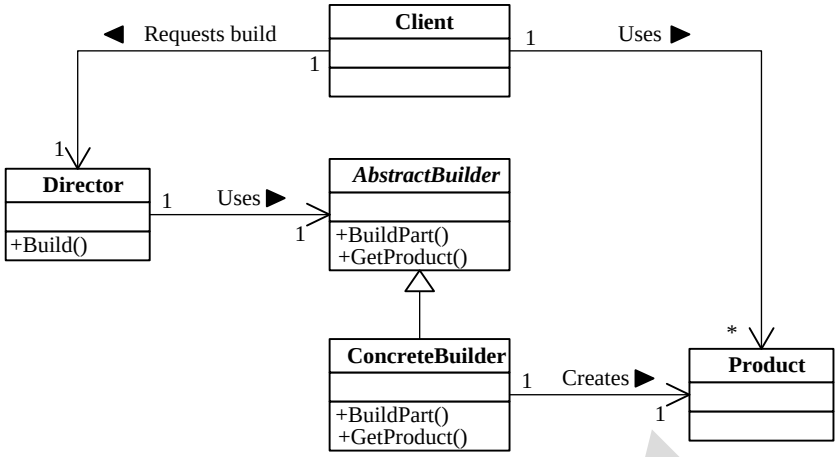

Figure 18.1 Class Diagram for Builder

The class diagram is shown in figure 18.1. The *Director* class parses the input data. The parsed data is sent to the *Builder* which then creates the product in a step-by-step fashion. This class has member functions for building the parts and for returning the finished product to client systems. A generic example of a sequence diagram showing the steps is given in figure 18.2.

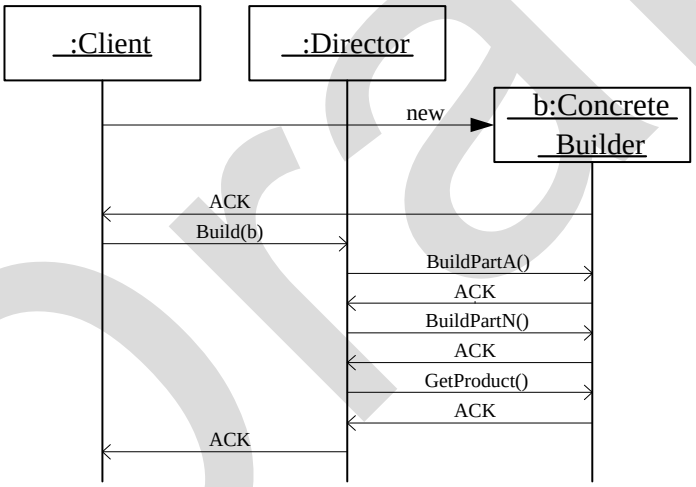

Figure 18.2 Sequence Diagram for Builder pattern

Focusing on the data flow issues instead of the more static class diagrams makes it easier to understand and apply this patten in software projects.
Typical candidates for the *Builder* pattern is the complex object network as shown in Figure 18.3 and this is the basis for a Monte Carlo software engine. The top-level object `MCEngine` is a whole-part object and its parts correspond to stochastic differential equations, finite difference methods and random number generators. In future versions the class network will need to be extended to support more requirements and non-object-oriented paradigms. The implementation of a given requirement implies the design of new C++ classes or modifications of existing classes that are then integrated in the class diagram in figure 18.3.

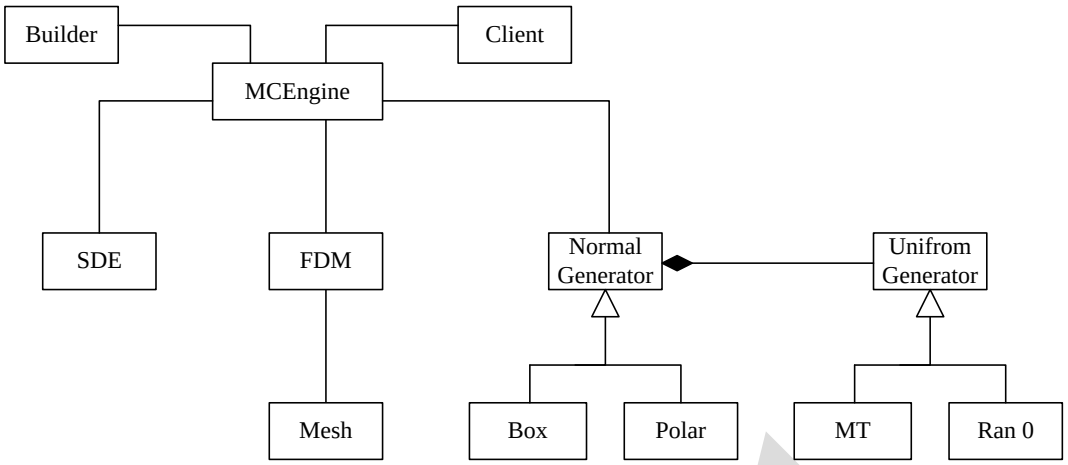

Figure 18.3 Class Diagram for Monte Carlo Framework

Some final remarks on the design and implementation of the classes in figure 18.3:

- We could design the builder object in such a way that it delegates parts of its creational activities to factory objects. In this case we use *sub-contractors* to create the parts of the `MCEngine` object. For example, we could use *subcontractor* factories to create SDE, FDM and random number generator objects.
- An important issue is to determine the structure and format of the input data that the *Director* needs to process. We could use enumerated types to distinguish between the different kinds of derived classes. In more complex applications we need to define a language to describe the input data and a parser to extract the tokens and building blocks that eventually form the parts of the finished product. To this end, the *Interpreter* pattern (GOF 1995) allows us to
  - defines a representation for the grammar of the language that we have chosen, and use the representation to interpret sentences in the language.

18.5 Structural Patterns

The GOF structural patterns are subsumed by the higher-level POSA patterns such as *PAC, Layers* and *Whole-Part* patterns. GOF structural patterns should partition objects into networks of dedicated objects in such a way as to satisfy the *Single Responsibility Principle (SRP)*.

Which kinds of object decomposition problems lead to the discovery of GOF structural patterns? Some scenarios are:

- Allowing an object to have several implementations or realisations.
- Create a unified interface to a collection of objects.
- Place a surrogate/proxy object between two objects.
- Create trees of objects and recursive aggregates.
- Convert the interface of a class into another interface that clients expect.

The discovery and implementation of structural patterns is crucial to the quality of software applications and it is for this reason that we introduce structural patterns followed by a discussion of creational and behavioural patterns.

18.5.1 Facade Pattern

This pattern is used to make a subsystem (consisting of a network of classes) easier to use. In general, the facade object provides a simple and unified interface to a collection of objects. The client communicates with the facade which in its turn delegates to its collaborator objects. There are two main scenarios when looking for facades:

- We discover them in the early stages of the software development process when we use the *Whole-Part* pattern, for example. Many GOF patterns are facades.
- We discover the need for them when client code interfaces with too many objects, resulting in code that becomes difficult to maintain. We then need to reduce the degree of coupling between objects.

In the second scenario we group objects in some way and we consider this to be a reengineering or *refactoring* process. This is an option when you start realising that your object network is becoming too complex and when corrective action needs to take place, sooner rather than later.

The *Facade* pattern is a general concept and many of the specific GOF patterns are instances of it. It is pervasive in software systems.

18.5.2 Layers Pattern

This pattern is discussed in detail in POSA 1996. A common use is when we model a PAC agent. In all cases we have decomposed an agent into three independent components corresponding to the data, the user interface and the control aspect. This initial separation of concerns will help us when we elaborate the design of these components using the GOF patterns. The two possible structural representations of a PAC agent are shown in Figure 18.4.

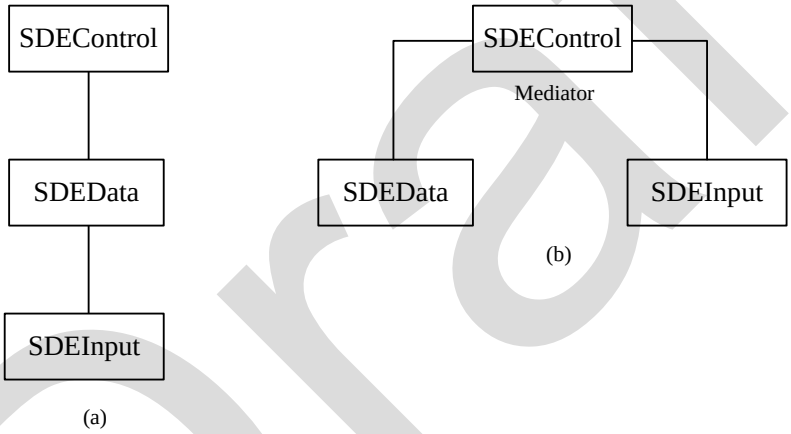

Figure *18.4 Layers pattern: (a) three-layer case; (b) two-layer case*

18.6 Behavioural Patterns

Once we have discovered the classes, class hierarchies and class relationships in the application we need to design their member functions. In particular, requirements evolve and code may need to be changed. Some scenarios are:

- S1: The body of a member function is replaced by new code.
- S2: Define a family of interchangeable algorithms that clients can use.
- S3: Extend the functionality of all classes in a class hierarchy in a non-intrusive way.
- S4: Promote common data and functionality (so-called *commonality*) from derived classes to a common base class.

Scenarios S1 and S2 are realised by the *Strategy* pattern (and sometimes by the *State* pattern); the *Visitor* pattern realises scenario S3 while the *Template Method* pattern is used to realise scenario S4. These patterns compete in certain contexts but they can also collaborate to form *pattern languages*. A *pattern language* is a structured method for describing good design practices within a field of expertise. It is characterised by:

- Noticing and naming the common problems in a field of interest.
- Describing key characteristics of effective solutions that meet some stated goal.
- Helping the designer move from problem to problem in a logical way.

- Allowing for many different paths through the design process.

We shall see some examples of these patterns in the following sections.

18.6.1 *Visitor* Pattern

We have given some examples of the application of *Visitor* to computational finance in Duffy 2004 and Duffy 2006. The focus was on partial differential equations and finite difference methods. We have already shown its applicability to the Monte Carlo applications in Duffy 2010. In general, we use this pattern when we extend the functionality of classes in a class (context) hierarchy or when we wish to create input and output functionality for these classes. This pattern addresses a fundamental design problem in software development, namely defining classes in a class hierarchy and subsequently defining new operations for these classes. However, we do not wish to implement these operations as member functions of the classes themselves because this increases code bloat and makes the classes more difficult to maintain. Instead, we create another class hierarchy and the classes in this hierarchy contain functions that implement the new functionality associated with the context classes. This is the intent of the *Visitor* pattern.

18.6.2 Strategy and Template Method Patterns

These are two related behavioural patterns and they are used when we create *algorithms*. In particular, these patterns allow the developer to design and implement algorithms as classes (usually they are part of a class hierarchy having standard interfaces). The body of the code that implements an algorithm is hidden in member functions. Furthermore, it is desirable to standardise the types of the input and output parameters of the algorithm. This leads to maintainable code.
We first discuss *Strategy*. This pattern allows us to define a family of algorithms by encapsulating each one in a class. We make the algorithms interchangeable by deriving the corresponding classes from a general abstract base class. The added value is that the algorithms and clients can vary independently, thus allowing the algorithms to become more reusable.
When designing strategy classes we can choose between an object-oriented approach (base and derived classes, as discussed in GOF 1995) or we can use *policy classes* and .NET delegates. We discuss the latter topic in section 18.8.

We now discuss the *Template Method Pattern*. It is similar to the *Strategy* pattern in that it models algorithms, but in contrast to Strategy - where the complete code body of an algorithm is replaced by other code - this pattern describes an algorithm as a series of steps, some of which are *invariant* (which means that the corresponding code does not need to be replaced by other code) and some of which is *variant* (it may need to be replaced by other code). In short, the algorithm has customisable (*variant*) and non-customisable (*invariant*) parts. The advantage is that we can replace variant code by other variant code while retaining the structure and the semantics of the original or 'main' algorithm.
How do we implement the *Template Method* pattern? The general idea is to define a base class B and one or more derived classes (call them D1, D2, ..). The tactic is as follows:

- Define the member function for the main algorithm in B.
- Define 'hook' (variant) functions as pure virtual functions in B.
- Implement these hook functions in D1 and D2.

Thus, this solution employs a combination of inheritance and polymorphism to implement the pattern.

18.7 Builder Application: Calibration Algorithms to Cap and Floor

In this section we discuss applying the *Builder* pattern to calculate caplet volatility matrices and volatility matrices for many strikes. We use the standard ingredients that are needed in this pattern and that we have already discussed in section 18.4. The example has been chosen for didactical reasons to show the steps that we apply in the *volatility calculation process*. The goal is to read market data (for example, discount factors, year fraction, tenor and flat cap volatility) and then to calculate caplet volatilities and finally to store them in a caplet

volatility matrix. There are many ways to build volatilities matrices; in our didactical example we examine the *iterative* and *best fit* models.
We have already discussed many of these issues in previous chapters.

18.7.1 Caplet Volatility Matrix

We discuss the *Builder* pattern to help us create caplet volatility matrices using a variety of building methods. The UML class diagram is shown in Figure 18.5. The classes are:

- `CapletVolMatrixBuilder`: the abstract class (or interface) that has abstract methods for the product parts as well as for the finished product.
- `BestFitBuilder` and `IterativeBuilder`: these are concrete builder classes. The former class uses an optimisation method by minimising the difference between the market premium and the recalculated market premium.
- `CapletVolMatrix`: this is the final product and it contains bootstrapped caplet volatilities. It also stores some important data used in the building process.

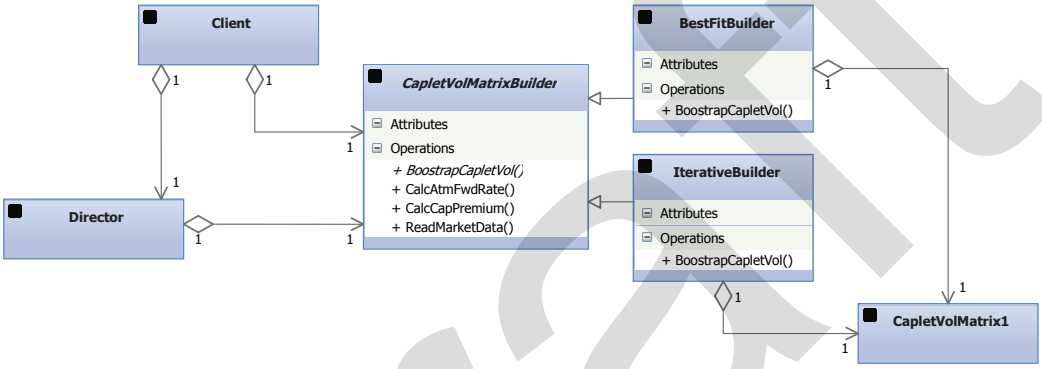

Figure 18.5 Builder Pattern for Cap and Floor

We execute a number of steps in order to build the final product:
1. Read market data.
2. Calculate the at-the-money (ATM) forward rate for each caplet.
3. Calculate the cap premium using data from steps 1 and 2.
4. Bootstrap caplet volatilities, compute caplet volatilities and store them in a matrix.

The steps 1, 2 and 3 are common to all builder implementations and hence the corresponding methods `ReadMarketData`, `CalcAtmFwdRate` and `CalcCapPremium` will be implemented in the base class `CapletVolMatrixBuilder`.

18.7.2 Volatility Matrix with multiple Strikes

We now consider an extended example as shown in Figure 18.6. In this case we calculate caplet volatilities for a single strike, for multiple strikes and for at-the-money (ATM) strikes. The main players are:

- `Director` : It constructs a `CapletVolMatrix` according to a sequence of operations. It needs instances of `DiscountCurve`, `MktParVol` and `FwdCurve` as input data. It calculates at the money forward rates. Finally, `ICapletVolMatrixBuilder` has a special way to build `CapletVolMatrix`.
- `ICapletVolMatrixBuider` : This is the interface for creating the object `CapletVolMatrix`.
- `MultiStrikeBuilder` : This class creates `CapletVolMatrix` for many strike. Basically it uses `MonoStrikeBuilder` many times.
- `MonoStrikeBuilder` : The class that creates `CapletVolMatrix` for only one strike.
- `AtmStrikeBuilder` : This class is derived from `MonoStrikeBuilder`. It should bootstrap caplet volatility of at the money strike. The technology is like a mono strike builder where the strike is the ATM strike for each caplet.

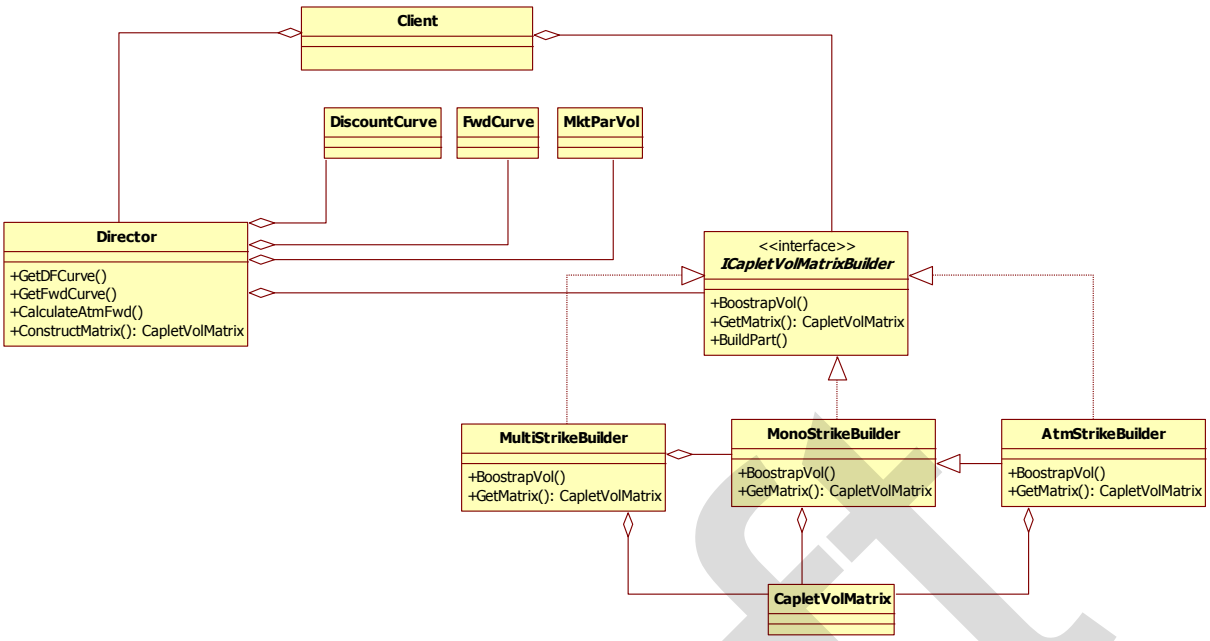

Figure 18.6 Extended Builder

The interfaces of these classes are shown in Figure 18.6 and this information in combination with the class structure can be used as input to an implementation of the *Builder* pattern in this case.

18.8 .NET Delegates

C# supports a number of advanced topics that promote the flexibility and reusability of C# software systems. In this section we introduce delegates. A *delegate* is a class that dynamically wires up a method to its target method. We must realise that it is based on principles that are different from the object-oriented and generic principles that we already have discussed in this book. First, a *delegate type* defines a *protocol* in the sense that it has input arguments and return type but no body. It may even have a name. These three elements are sometimes called the *signature* of the delegate type. We cannot call the type because it has no body. It is important to remember that all code using a delegate must conform to the signature. Next, a *delegate instance* is an object that refers to one (or more) target methods conforming to the protocol.
We take the first example on how to define and use a delegate. The example simulates choosing an algorithm that executes a certain (simple) algorithm. The delegate type is defined as:

```
// Delegate type
delegate double Compute(double x);
```

We now have the freedom to define several delegate instances that conform to the delegate's protocol, for example:

```
// Delegate instances
static double MyExp(double x) { return Math.Exp(x);}
static double MyLog(double x) { return Math.Log(x);}
static double MySquare(double x) { return x*x;}
```

Finally, we can use these instances at run-time, as following code shows:

```
// Selecting a specific delegate
int choice = 1;
Compute t = MyLog;
if (choice == 1)
     t = MySquare;
```

```
double x = 60.0;
Console.WriteLine("Computed value: {0} ", t(x));
t = MyLog;
Console.WriteLine("Computed value: {0} ", t(x));
```

This example shows the essence of what delegates are and how to use them. In this particular case we use them to implement algorithms, similar to how the *Strategy* pattern works. In fact, we can replace the GOF *Strategy* pattern based on the object-oriented paradigm by one that is based on delegates.
Finally, we remark that invoking a delegate is just like invoking a normal method. For example, we can create and invoke a delegate by using the somewhat verbose code:

```
Compute t2 = new Compute(MyExp);
double value = t2.Invoke(1.0);
Console.WriteLine("Computed value: {0} ", value);
```

For completeness, we show the full code as one unit:

```
// Delegate type
delegate double Compute(double x);

public class Delegate101
{
     static void Main()
     {

   // Selecting a specific delegate
   int choice = 1;
   Compute t = MyLog;
   if (choice == 1)
                            t = MySquare;

   double x = 60.0;
   Console.WriteLine("Computed value: {0} ", t(x));
   t = MyLog;
   Console.WriteLine("Computed value: {0} ", t(x));

   Compute t2 = new Compute(MyExp);
   double value = t2.Invoke(1.0);
   Console.WriteLine("Computed value: {0} ", value);

   }

   // Delegate instances
   static double MyExp(double x) { return Math.Exp(x);}
   static double MyLog(double x) { return Math.Log(x);}
   static double MySquare(double x) { return x*x;}

}
```

We now show how to create code that applies a delegate to a collection. To this end, we wish to modify the elements of an array using the already defined `Compute` delegate:

```
static void Transform(double[] values, Compute t)
{
     for (int j = 0; j < values.Length; ++j)
     {
             values[j] = t(values[j]);
     }
}
```

An example of use is:

```
// Apply a delegate to a collection
double[] values = { 10.0, -20.0, 5.0, 9.7 };
Transform(values, MyExp);
```

In this case is it possible to create a generic version of this code and thus build a small library of useful functionality.
We now elaborate on how to use delegates in more advanced situations.

18.8.1 *Provides* and *Requires* Interfaces: creating Plug-in Methods with Delegates

C# is a popular language and it allows developers to create flexible applications by encapsulating domain entities in classes. Furthermore, we can create complex classes from simpler ones using the *Composition*, *Aggregation* and *Inheritance* mechanisms. In general, these are client-server relationships between one class (the *client*) that uses the services of one or more other classes (called *server* classes) by calling their member functions. This situation leads to an O*bject Connection Architecture* (OCA) because all inter-module connections are from object to object. The major disadvantage is that all modules and classes must be built before the architecture is defined and hence this approach cannot be used to lay out the plan for a software system. This is in contrast to an *Interface Connection Architecture* (ICA) that defines all connections between components of a system using only interfaces. Interfaces need to specify both *provided* and *required* features. In general, a *feature* is a computational element of a component, for example a function, port or action. For a detailed introduction to object and interface connection architectures, see Leavens 2000.

The crucial issue is to implement provides and requires interfaces. To this end, we use C# delegates. Before we go into the details we describe these interfaces using standard UML *component diagrams*, an example of which is shown in Figure 18.7. In this case component C1 provides the interface I3 to potential clients and it has the requires interfaces I1 and I2 from server components C2 and C3, respectively. In other words, C1 offers services but it also requires services from other server components.

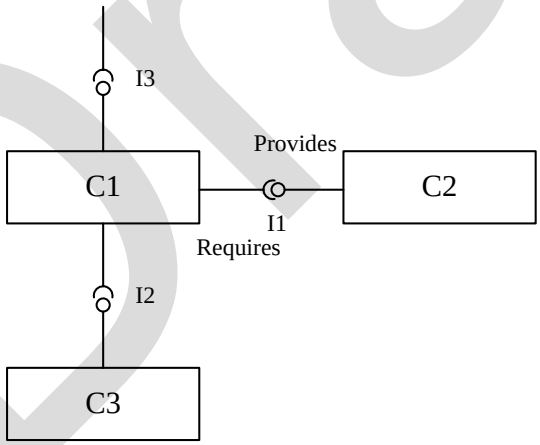

Figure 187 UML Component Diagram

We now give a simple example of a class that implements the price function for the Black Scholes formula and that shows how to implement provides and requires interfaces. This class also requires the data from another interface that we implement as a delegate. This latter entity is responsible for producing the actual data that is needed by the pricing formula. The main objective is to show how to price a call option by implementing the features similar to what we see in Figure 18.7. In particular the client class `Pricer3` provides an interface to compute the option price. It communicates with an object that is responsible for creating the data that is used by the pricing formula. The client has no knowledge of the precise implementation of the data source object; this decision has been *delayed* and it for other parts of the software system to implement it:

```
public struct Pricer3      // One version of an implementation of ICA
{ // A class that offers an interface and requires another interface
```

```
   public delegate void DataSource(ref Data data);
   public DataSource ds;

   public double compute(double S)
   {
     // Define the data and slot
     Data data = new Data();

     // Connect to slot and initialise the data
     ds(ref data);

     double tmp = data.sig * Math.Sqrt(data.T);

     double d1 = ( Math.Log(S/data.K)
            + (data.b+ (data.sig*data.sig)*0.5 ) * data.T )/ tmp;
     double d2 = d1 - tmp;

     return (S * Math.Exp((data.b-data.r)*data.T)
            * SpecialFunctions.N(d1))
            - (data.K * Math.Exp(-data.r * data.T)
            * SpecialFunctions.N(d2));
   }
```

We apply delegates to load default specifications. This functionality can also be realised using the *Prototype* pattern. In this sense we use delegates to initialise data and we can then see as being a creational pattern.

```
public static void PlainDataSource(ref Data val)
{ // Simple data source; standard stock

   val.T = 0.25;
   val.K = 65.0;
   val.r = 0.08;
   val.sig = 0.3;
   val.b = val.r;
}
```

We now customise this class in different ways. For example, we can implement the data source as a struct as the following code shows:

```
public struct Data
{ // Option data

   public double T;
   public double K;
   public double r;
   public double sig;
   public double b; // Cost of carry
}

/* a) Black-Scholes (1973) stock option model: b = r
   b) b = r - q Merton (1973) stock option model with continuous dividend yield
   c) b = 0 Black (1976) futures option model
   d) b = r - rf Garman and Kohlhagen (1983) currency option model, where rf is the
   'foreign' interest rate
*/


public enum OptionType {Stock, Index, Future};

public struct GeneralisedDataSource
{ // Allows for different kinds of options; this is a function object

   public OptionType optType;
```

```
   public GeneralisedDataSource(OptionType optionType)
     { optType = optionType; }
   public void init(ref Data val)
   {
     val.T = 0.25;
     if (optType == OptionType.Future)
            val.b = 0.0;

     // more options

     val.K = 65.0;
     val.r = 0.08;
     val.sig = 0.3;
   }
}
```

We now use these functions in a test program which we call the *major client* because it is here that we decide to use these functions. The class `Pricer3` knows nothing about these functions and is *policy-free* in this sense:

```
public class Test_ICA
{
   static void Main()
   {
     {
     Pricer3 pricer = new Pricer3();
     pricer.ds = Pricer3.PlainDataSource;

     double S = 60.0;
     Console.WriteLine("Stock,generalised version:{0} ", pricer.compute(S));
     }

     {
     GeneralisedDataSource mySource
            = new GeneralisedDataSource(OptionType.Future);
     Pricer3 pricer = new Pricer3();
     pricer.ds = mySource.init;

     double S = 60.0;
     Console.WriteLine("Stock, full generalised version: {0} ", pricer.compute(S));
     }

   }
}
```

18.8.2 Multicast Delegates

We now discuss how to define a delegate instance that is able to reference *multiple target methods*. In other words, it can trigger multiple target methods and in this sense it is called a *multicast delegate*. The concept is similar to multicasting in computer networking. To this end, C# uses the operators + and += to combine delegate instances. Furthermore, we can use the operators – and -= to remove one delegate instance from another one. We take an example of a simple calculator that consists of algorithms that take two scalar input arguments and that produces a scalar value as output. To show some variation, we propose two equivalent protocols:

```
public delegate void Compute(double v1, double v2, out double answer);
public delegate double ComputeII(double v1, double v2);
```

Next, we create a class whose methods conform to the above signatures:

```
public class Calculator
{

   // Methods returning void
   public static void Add(double v1, double v2, out double answer)
```

```
    {
      answer = v1+v2;
      Console.WriteLine("Add {0}", answer);
    }

    public static void Subtract(double v1, double v2, out double answer)
    {
      answer = v1-v2;
      Console.WriteLine("Subtract {0}", answer);
    }

    // Methods returning double
    public static double Multiply(double v1, double v2)
    {
      return v1*v2;
    }

    public static double Divide(double v1, double v2)
    {
      return v1/v2;
    }
}
```

We now create some multicast delegate instances and we compute them as follows:

```
class TestMulticastDelegate
{

    static void Main()
    {
      // Signature type I
      Compute generator = Calculator.Add;
      generator += Calculator.Subtract;

      double a = 4.0; double b = 2.0; double c;
      generator(a, b, out c); // Will print Add: 6 and Subtract: 2

      generator -= Calculator.Subtract;
      generator(a, b, out c); // Will print Add: 6

      // Signature type II
      ComputeII generatorII = Calculator.Divide;
      generatorII += Calculator.Multiply;

      double x = 4.0; double y = 2.0;
      Console.WriteLine("Generator II: {0}",generatorII(x, y));          // 8

      generatorII -= Calculator.Multiply;
      Console.WriteLine("Generator II: {0}", generatorII(x, y));         // 2
    }
}
```

We can generalise this approach to more general applications, in particular applications in which the traditional GOF *Observer* have been used for example when logging different kinds of data at specific time points in a Monte Carlo simulation.

18.8.3 Generic Delegate Types

A delegate type may contain generic type parameters. In other words, it is possible to define delegate types whose input arguments and/or return types are generic. This means that we can write generic code once and reuse it by instantiating its generic parameters. As an example, we create a generic version of the original example in section 18.8. The new generic delegate type is:

```
// Generic Delegate type
```

```
delegate T Compute<T>(T x);
```

We can then create a method that transforms a generic collection:

```
static void Transform<T>(T[] values, Compute<T> t)
{
   for (int j = 0; j < values.Length; ++j)
   {
     values[j] = t(values[j]); Console.WriteLine("{0}", values[j]);
   }
}
```

We can now call this method for any specific data types:

```
// Delegate instances
static int MySquare(int x) { return x*x;}
static long MySquare(long x) { return x*x; }

// Apply a delegate to a collection
int[] values = { 1, -2, 3, 4 };
Transform<int>(values, MySquare);

long[] valuesB = { -1, -2, -3, -4 };
Transform<long>(valuesB, MySquare);
```

Continuing, .NET provides the developer with a number of commonly needed function types to model functions and subroutines (known as *actions*), for example:

```
// Generic lambda expressions
delegate T FuncOne<T>(T t);
delegate T Func<T>(T t1, T t2);
delegate void Action<T>(T t1, T t2);
```

We now instantiate these delegates using lambda functions:

```
FuncOne<int> f1 = (int x) => x * x;
Console.WriteLine(f1(3));

FuncOne<double> f2 = (double x) => x * x;
Console.WriteLine(f2(3.1415));

Func<int> f3 = (int x, int y) => x + y;
Console.WriteLine(f3(3, 4));
```

18.8.4 Delegates versus Interfaces

Problems that delegates solve can also be solved using interfaces. We would use delegates instead of interfaces if one or more of the following conditions are met:

- The interface only needs a single method.
- We need multcast capability.
- The user/observer needs to implement the method several times.

In general, delegates lead to more loosely-coupled code than equivalent code that uses interfaces. Finally, it is possible to aggregate or bundle delegates in a class to form a *plug-and-socket* components.

18.9 The Standard Event Pattern in .NET and the Observer Pattern

Delegates are close in spirit to the GOF *Observer* pattern because both mechanisms are concerned with event notification between a *broadcaster* (also known as *publisher*, *observable* or *subject*) and a *subscriber* (also known as *observer*). The broadcaster contains a delegate instance as member data. Events can take place in the

broadcaster and it then invokes its embedded delegate. Subscribers are the recipients of target methods. They can *subscribe* and *unsubscribe* to a broadcaster by calling += and -=, respectively. The .NET *Event* pattern is a language-dependent implementation of the *Observer* pattern. An *event* is a language construct that exposes a subset of delegate functionality that we need for this pattern. To this end, the *Event* pattern ensures that subscribers do not interfere with each other.

The .NET Framework defines a standard pattern to support event modelling. First, we use the class `System.EventArgs` which is a base class for conveying information about an event. Users create derived classes in order to convey old and new values of some quantity of interest.
We take an example to show how the pattern works. The main steps are:

- Define a derived class of `EventArgs` to hold changeable data.
- Define an event of the desired delegate type in the subscriber.
- Create a protected virtual method that fires the event.

The current derived class of `EventArgs` is given by:

```
public class CoordinateChangeEventArgs : System.EventArgs
{
   public readonly double val;

   public CoordinateChangeEventArgs(double value)
   {
            val = value;
   }
}
```

This class exposes data as readonly data. The next step is to define the delegate for the event. It must have a `void` return type and it has two input arguments; the first argument is of type `object` and the second argument corresponds to a derived class of `EventArgs`. In other words, the first argument is the event broadcaster and the second argument is the data to convey. Finally, the name of the delegate must end with `EventHandler`. To this end, .NET defines the following generic delegate:

```
public delegate void EventHandler<TEventArgs>
   (object sender, TEventArgs e> where TEventArgs : EventArgs;
```

We now define an event of the desired type in the broadcaster:

```
public class Observable
{

   double x; double y;
   public Observable(double X, double Y) { x = X; y = Y; }

   // Delegate
   public event EventHandler<CoordinateChangeEventArgs> coordChanged;

   // Method to fire the event
   protected virtual void OnCoordChanged(CoordinateChangeEventArgs e)
   {
     if (coordChanged != null)
     {
            coordChanged(this, e);
     }
     else
     {
            Console.WriteLine("No change, no observers");
     }

   }
```

```
    public double X
    {
      get { return x;}
      set
      {
              x = value;
              OnCoordChanged(new CoordinateChangeEventArgs(value));
      }
    }

    public double Y
    {
      get { return y;}
      set
      {
              y = value;
              OnCoordChanged(new CoordinateChangeEventArgs(value));
      }
    }

    public void print()
    {
      Console.WriteLine("Point: {0} {1}", x, y);
    }

}
```

We now create a program to test the pattern:

```
public class EventPattern
{

    static void Main()
    {
      Observable myObs = new Observable(1.0, 2.0);

      myObs.X = 99.0;
      myObs.Y = 88.0;
      myObs.print();

      // Attached observer
      myObs.coordChanged += CoordChanged;
      myObs.X = 32.0;
      myObs.Y = 44.0;
      myObs.print();

      // No attached observer
      myObs.coordChanged -= CoordChanged;
      myObs.X = 32.0;
      myObs.Y = 44.0;
      myObs.print();
    }

    static void CoordChanged(object sender, CoordinateChangeEventArgs e)
    {
      Console.WriteLine ("Change has occurred: ", e.val);
    }
 }
```

The output is now:

```
No change, no observers
No change, no observers
Point: 99 88
Change has occurred:
```

```
Change has occurred:
Point: 32 44
No change, no observers
No change, no observers
Point: 32 44
```

We have now completed our discussion of .NET events. They are used extensively in *WinForms*, for example when a button mouse click event is handled by some class. We summarise the steps when using the *Event* pattern:

1. Create an event class derived from `EventArgs`.
2. Observable class: define event delegate; define event variable to store subscribers; call event variable when an event fires; event variable calls all subscriber methods.
3. Subscriber class: create a method that implements an event delegate; subscribe to the observable by adding this method to the observable's event variable.

We conclude this section with an example of a simple clock that is continuously updated. The event arguments class is:

```
public class TimeChangeEventArgs: EventArgs
{
   public DateTime dt;

   public TimeChangeEventArgs(DateTime dt)
   { // Constructor

     this.dt=dt;
   }
}
```

The observable class is:

```
public class Clock
{
   // Define event delegate
   public delegate void TimeChangeEventHandler (object sender, TimeChangeEventArgs e);

   // Event variable to store subscribers. Note, it also works without the
   // "event" keyword but with "event" the framework can make a difference.
   public event TimeChangeEventHandler OnTimeChange;

   public void Run()
   {
   // Infinite loop, sleeps every iteration for 1000 ms
   for (;;Thread.Sleep(1000))
   {
     // Get the current time
     TimeChangeEventArgs args
            = new TimeChangeEventArgs(DateTime.Now);

     // Raise event and call event methods;check for null
     if (OnTimeChange!=null) OnTimeChange(this, args);
   }
   }
}
```

Finally, the code for subscribers and test program is:

```
public class ClockSubscriber
{
   public static void Main()
   {
     // Create clock instance
```

```
        Clock clock=new Clock();

        // Subscribe event handlers for Clock.TimeChangeEvent
        clock.OnTimeChange
                += new Clock.TimeChangeEventHandler(DisplayTime1);
        clock.OnTimeChange
                += new Clock.TimeChangeEventHandler(DisplayTime2);

        // Start the clock
        clock.Run();
      }

      private static void DisplayTime1(object sender, TimeChangeEventArgs args)
      { // TimeChangeEventHandler 1

        Console.WriteLine("DisplayTime 1: {0}", args.dt);
      }

      private static void DisplayTime2(object sender, TimeChangeEventArgs args)
      { // TimeChangeEventHandler 2

        Console.WriteLine("DisplayTime 2: {0}", args.dt);
      }
  }
```

18.10 A PDE/FDM Patterns-based Framework for Equity Options

In this section we discuss an object-oriented framework based on GOF patterns that we have already discussed in section 18.2 in a simpler form. We reduce the scope to one-factor linear partial differential equations that describe the behaviour of equity (and fixed income) options on a single underlying variable. The main goal is to create a software framework that we can adapt and extend to suit a range of requirements based on the assumption that we are modelling options using a PDE approach, for example:

- R1 The ability to model parabolic PDEs in different forms, for example:

$$\begin{aligned}
&\frac{\partial u}{\partial t} = a\frac{\partial^2 u}{\partial x^2} + b\frac{\partial u}{\partial x} + cu + f \text{ (non-conservative)}.\\
&\frac{\partial u}{\partial t} = a\frac{\partial}{\partial x}\left(b\frac{\partial u}{\partial x}\right) + c\frac{\partial u}{\partial x} + du + f \text{ (conservative)}.\\
&\frac{\partial u}{\partial t} = a\frac{\partial}{\partial x}\left(b\frac{\partial u}{\partial x}\right) + cu + f \text{ (conservative diffusion)}.\\
&\frac{\partial u}{\partial t} = a\frac{\partial^2}{\partial x^2}(bu) + c\frac{\partial}{\partial x}(du) + eu + f \text{ (Fokker-Planck)}.
\end{aligned} \tag{18.1}$$

- R2 The domain of integration in the underlying variable x can be an infinite interval, a semi-infinite interval or a bounded interval. It must be possible to transform a PDE on one interval to a PDE on another interval.
- R3 It must be possible to support a wide range of diffusion and drift coefficients in the PDEs, boundary conditions and payoff functions. In particular, the framework should support discontinuous coefficients and coefficients that are generated from a calibration module.
- R4 Support for both plain and early exercise options as well as barrier and lookback options.
- R5 Support for continuous and discrete monitoring.
- R6 Calculation of option sensitivities, for example delta and gamma.
- R7 The data needed to initialise PDE data can originate from various sources.
- R8 Support for nonlinear PDEs.

These are the main requirements pertaining to the PDEs that we are interested in approximating using the finite difference method. Since much effort goes into creating, testing and debugging finite difference schemes to approximate the solutions of PDEs it is obvious that the amount of coupling between them should be kept to a minimum. In particular, the following requirements are important:

- R9 The finite difference code should be strongly uncoupled from the code that implements the PDEs.
- R10 We need to support a range of finite difference solvers that approximate the solutions of equations (18.1) with a given accuracy and performance.
- R11 It must be possible to add new PDEs and finite difference solvers to the framework in order to allow quants and model validators to create new models and test existing models.
- The methods and designs used for one-factor problems should be generalisable to n-factor problems. In particular, it should be possible to adapt the design to accommodate Asian, multi-asset, Heston and SABR models, for example.

Based on these requirements, we discuss how to design and implement finite difference solvers for one-factor PDE option pricing models.

18.10.1 High-Level Design

In this section we discuss the steps to analyse, design and implement finite difference schemes in C#. Instead of jumping directly into code and suffering the maintainability consequences we decide to analyse the problem from a number of orthogonal viewpoints that are used in software engineering projects:

- *Dynamic viewpoint*: finite difference methods implement some kind of one-step or multi-step marching algorithm in time (from time zero to the expiry time T). We wish to know what happens at each discrete time level, for example which data structures are being updated and which constraints need to be satisfied. The de-facto standard modelling technique is based on *UML Statecharts* in which we model the system as a sequence of *states*. *Transitions* bring the system from one state to another one and *actions* are functions that are triggered when a transition fires.
- *Data viewpoint*: this model describes the data in the system, what the data structures are and how they are updated. In the current one-factor case we use a matrix to store the option price at each discrete time step for a range of values of the underlying variable. Each row of this matrix corresponds to a range of values of the underlying variable at a given time level. In an object-oriented setting the data in this viewpoint will be member data of some class (usually a mediator) in the software system.
- *Functional viewpoint*: this model describes how the data in the data viewpoint is updated. In an object-oriented setting the functions in this viewpoint correspond to the member functions of the classes in the data viewpoint and they are thus responsible for accessing the data. Furthermore, the functions in this model correspond to *transitions* and *actions* in the dynamic viewpoint model.

Summarising, these three viewpoints allow us to analyse a problem from three orthogonal viewpoint. First, the data model describes the ‘what’, the functional model describes ‘how’ the data is accessed and finally the dynamic model describes ‘when’ the data is accessed. Once we understand these viewpoint then design and implementation will be easier.

We now discuss how we document each of these models in the current context. We first discuss the data model. The main data structure is a matrix as already mentioned and since we are employing one-step finite difference methods we create two vectors to hold option values at time levels n and n + 1. We discuss the data model first because it offers a tangible starting point for the analysis. To this end, we define the following collections:

```
// Solutions at time levels n and n+1
protected Vector<double> vecOld;
public Vector<double> vecNew;

// Results at ALL levels
private NumericMatrix<double> res;
```

In the current implementation we assume constant mesh sizes in both the underlying and time directions. First, we create mesh arrays of given sizes `J` and `N`, respectively:

```
// Mesh arrays
protected Vector<double> xarr;
```

```
protected Vector<double> tarr;
```

These arrays are initialised by delegating to a class that generates meshes:

```
// Allow range[ A, B ] in x direction and [t1, T] in t  direction; create mesh arrays
Mesher m = new Mesher(xaxis.low,xaxis.high, taxis.low, taxis.high);
xarr = m.xarr(J);
tarr = m.yarr(N);
```

We are now ready to initialise the data structures that will contain option prices:

```
// The 3 data structures should be 'compatible' with each other, indices
vecOld = new Vector<double>(xarr.Size, xarr.MinIndex);
vecNew = new Vector<double>(xarr.Size, xarr.MinIndex);

res = new NumericMatrix<double>(tarr.Size, xarr.Size,
          tarr.MinIndex, xarr.MinIndex);
```

The next stage is to describe the *lifecycle* of this data and for this we employ statechart as shown in Figure 18.8 (in fact, it is a *State* pattern in the sense of GOF 1995). It consists of a number of states that correspond to data initialisation, data updating and postprocessing. Associated with each state are:

- *Entry actions*: the functions that are called when the state is entered.
- *Do (activity)*: the function or algorithm that is executed while in the given state.
- *Exit actions*: the functions that are called when the state is exited.

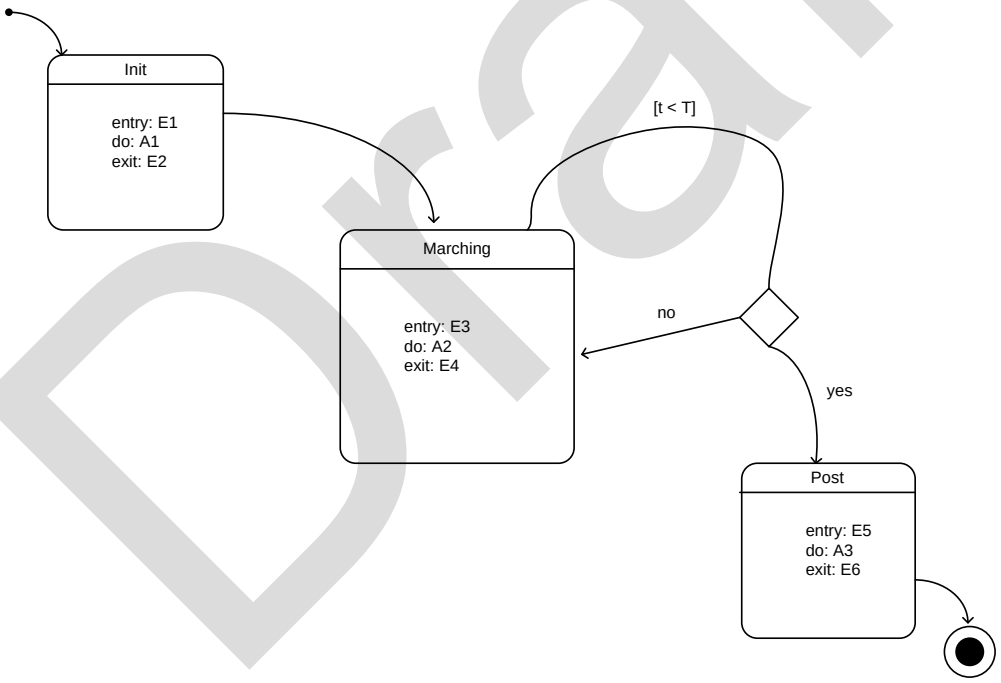

Figure 18.8 State Machine of FDM application

In general, we specify all entry actions, exit actions and activities for each state in a statechart before we implement it in C#. The advantage of creating a statechart for a problem is that we are forced (and are able) to analyse and think hard about the problem before we start implementing it, thus avoiding rework and ad-hoc programming. In the current version of the software the state machine has been implemented as follows:

```
public void initIC()
{ // Utility function to initialise the payoff function

   // Initialise at the boundaries
   vecOld[vecOld.MinIndex] = pde.BCL(taxis.low);
```

```
    vecOld[vecOld.MaxIndex] = pde.BCR(taxis.high);

    // Now initialise values in interior of interval using
    // the initial function 'IC' from the PDE
    for(int j = xarr.MinIndex+1; j <= xarr.MaxIndex - 1; j++)
    {
      vecOld[j] = pde.IC(xarr[j]);
    }

    // Matrix: rows are the time t, columns are the space x.
    res.Row(res.MinRowIndex, vecOld);
  }
```

and

```
  public NumericMatrix<double> result()
  {

    // The state machine; we march from t = 0 to t = T.
    for (int n = tarr.MinIndex+1; n <= tarr.MaxIndex; n++)
    {
      tnow = tarr[n]; // Next time level

      // The two methods that represent the variant parts
      // of the Template Method Pattern.
      calculateBC();         // Calculate the solution on the boundary
      calculate();           // Compute the solution at the new time level n+1

      // Add the current solution to the matrix of results.
      res.setRow(vecNew, n);

      tprev = tnow;
      for (int j = vecNew.MinIndex; j <= vecNew.MaxIndex; j++)
      { // Update value at time level n

              vecOld[j] = vecNew[j];
      }
    }

    return res;
  }
```

We now turn our attention to the functional model and to this end we have decomposed the system into a set of loosely-coupled subsystems in which each subsystem had a single major responsibility. The communication between subsystems can be realised by means of *provides-requires interfaces* I1, I2, ..., I5 as shown in Figure 18.9. Each subsystem's interfaces can be implemented by any of the following options:
A) C# interfaces.
B) Using classes with abstract and non-abstract methods.
C) Using delegates (with the possibility to use them as *callback functions*).

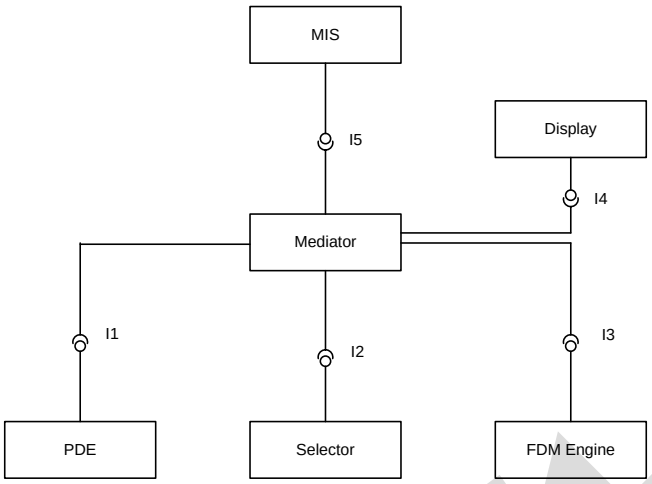

Figure 18.9 Top-level Component Diagram

In this book we have chosen for option B and this entails creating hierarchies of communicating classes. Then we can apply the GOF design patterns to allow us to create flexible and extendible software.

18.10.2 Detailed Design

We have created a number of version of the finite difference engine for one-factor problems in the past (see Duffy 2004, Duffy 2006) and we have produced several working versions. This fact means that we can improve on the original designs by incrementally modifying them. The current design is shown in Figure 18.10 and it is an UML class diagram. We discuss 'clusters of classes' where each cluster is assigned a letter for referencing purposes:

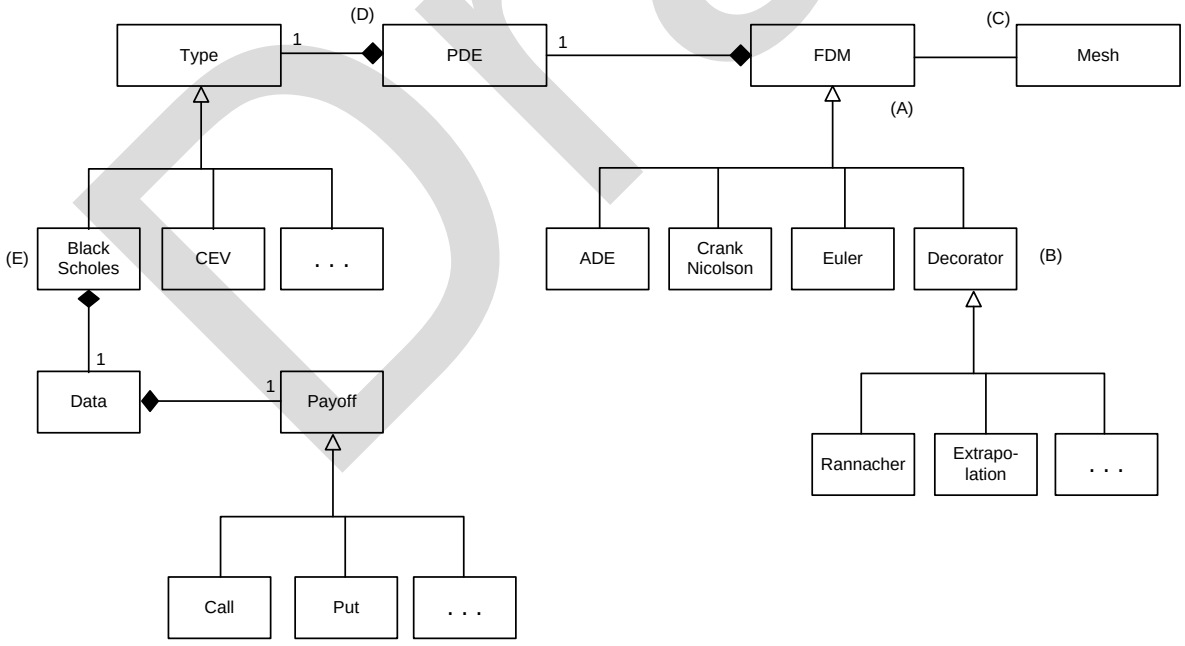

Figure 18.10 Class Diagram

- Cluster A: This is the hierarchy of classes that implements a range of finite difference schemes for one-factor finite difference schemes. We have discussed the Alternating Direction Explicit (ADE) method in Chapter 10 and we have used the *Template Method Pattern* that allows us to extend the hierarchy to other well-known schemes such as Crank-Nicolson, for example.

- Cluster B: We use the *Decorator* pattern to modify existing methods to suit certain needs. For example, we can use Richardson extrapolation in time to increase the accuracy of a scheme from first order to second order or from second order to fourth order depending on the accuracy of the basic finite difference method that is being decorated. Another application of the pattern is to create a finite difference method that combines the implicit Euler and Crank Nicolson methods (this is called the *Rannacher method*).
- Cluster C: This is the class (or subsystem) that generates uniform and adaptive meshes in the time and underlying directions. This class is reusable black box and can be implemented as *Strategy* pattern because it implements families of interchangeable algorithm families. Future versions will be more sophisticated.
- Cluster D: This is one class in the *Bridge* pattern. The class models one-factor partial differential equations including their coefficients, boundary conditions and initial condition. Part of its interface is:

```
class IBVP
{ // Pde == Initial Boundary Value Problem

   // The interface to use
   private Range<double> xaxis;
   private Range<double> taxis;
   private IIBVPImp imp                    // The implementation


   // ...

   public double diffusion( double x, double t )
   {
     return imp.diffusion( x, t );
   }

   // ...

   public double BCL( double t )
   {
     return imp.BCL( t );
   }

   public double BCR( double t )
   {
     return imp.BCR( t );
   }

   public double IC( double x )
   {
     return imp.IC( x );
   }

   double Constraint(double x)
   { // Test in American put option

     return imp.Constraint(x);
   }

}
```

- Cluster E: The classes that implement the application-independent classes in cluster D. We can accommodate a wide range of models in computational finance, for example Black Scholes, CEV, the heat equation and others. An example in the case of the Black Scholes equation is:

```
public class BSIBVPImp : IIBVPImp
{
   private Option m_option;

   public BSIBVPImp( Option option )
   {
     m_option = new Option( option );
```

```
    }


    // Coefficient of second derivative
    public double diffusion( double x, double t )
    {
      //  return 1.0;
              double v = m_option.Volatility;
              return 0.5 * v * v * x * x;
    }

    // ...

    // Left hand boundary condition
    public double BCL( double t )
    {
      // ...
    }


    // Right hand boundary condition
    public double BCR( double t )
    {
      // ...
    }

    // Initial condition
    public double IC( double x )
    {
      return m_option.PayOff( x );

    }

    public double Constraint(double x)
    { // Test in American put option

      return m_option.StrikePrice - x;
    }
}
```

- Cluster F: this is the subsystem whose responsibility is to create the data for the objects in cluster E. (The classes in cluster F are not shown in Figure 18.10). Creational patterns such as *Factory method*, *Prototype*, *Abstract Factory* are used to create the data while the *Strategy* pattern supports flexibility in the choice of payoff functions.

Finally, we can use the *Builder* pattern to construct the object network in Figure 18.10.

18.10.3 C# Code for Finite Difference Method

We now complete our discussion of the finite difference method and its implementation. We document the steps in the code:

```
class BSTestMain
{

    public static void Main()
    {

      // 1. Create an option using the Factory Method pattern.
      Option myOption = new OptionConsoleFactory().CreateOption();

      // 2. Define the pde of concern.
      IIBVPImp pde = new BSIBVPImp(myOption);
```

```
        // 3. Discrete mesh sizes.
        int J = 100;
        int N = J;


        // 4. The domain in which the PDE is defined.
        Range<double> rangeX
                       = new Range<double>(0.0, myOption.FarFieldCondition);
        Range<double> rangeT
                       = new Range<double>(0.0, myOption.ExspiryDate);

        // 5. Create FDM Solver.
        IBVPFDM fdm = new ADE(pde, rangeX, rangeT, J, N);

        // 6. Calculate the matrix result.
        NumericMatrix<double> sol = fdm.result();

        // 7. Display the results in Excel.
        ExcelMechanisms exl = new ExcelMechanisms();

        try
        {
                exl.printOneExcel(fdm.XValues, fdm.vecNew,
                        "ADE, T = 1/16", "NX=100", "NY=100", ",");
                exl.printOneExcel(fdm.XValues, exact, "Exact", "Col", ",", ",");
        }
                catch (Exception e)
        {
                Console.WriteLine(e);
        }
      }
}
```

The full source code is on the distribution medium.

18.10.4 Generalisations and Extensions

The methods, patterns and structures for the finite difference solver that we have introduced in this chapter can be reused in many kinds of applications. For example, the model in Figure 18.10 is based on the work in Duffy 2004 and the corresponding application is a special case of a Resource Allocation and Tracking (RAT) domain category. In other words, we can apply the patterns to a range of applications in computational finance on the one hand and we can extend the functionality of existing applications on the other hand. Of course, the names and responsibilities of the subsystems will be different in new applications but their number and the in the former case, we can apply the patterns to the following kinds of applications:

- The design of the application in Chapter 10 can be generalised by upgrading it to one based on Figure 18.10. In this case we can create more flexible software than was possible in Chapter 10.
- Monte Carlo engines are special cases of the RAT category. In this case the basic underlying subsystem models Stochastic Differential Equations (SDE) which are then approximated by appropriate finite difference schemes. One of the subsystems will be a generator of random numbers.
- Multi-factor PDE-based solvers: In this case the design for the one-factor model is used as springboard for new designs as the structures of both applications are similar.
- Support for a wider range of financial PDEs, for example, nonlinear Uncertain Volatility models, Fokker-Planck equations and PDEs in conservative form. For these applications we use the *Visitor* pattern to encapsulate code for specific finite difference schemes.

We can implement patterns using the object-oriented style as discussed in GOF 1995 or by the use of the *Delegates* mechanisms that we discussed in sections 18.8 and 18.9. They resolve man of the shortcomings of the traditional object-oriented programming model.

18.11 Summary and Conclusions

We have given an overview of the GOF patterns (GOF 1995), what they are and the kinds of common design problems that they solve. We then discuss how to apply the *Builder* pattern to create caplet volatility matrices and related calibration algorithms for single and multiple strikes. We also discuss a high-level software framework based on *Domain Architectures* (Duffy 2004) that we can progressively decompose until we get to the stage that we can discover and apply GOF patterns. We then have the option to implement these patterns using the traditional object-oriented programming style or using the .NET *Delegates* mechanisms that allows us to create flexible software systems. To this end, we have devoted a major section to this topic. In general, each GOF pattern can be implemented using delegates and with much less effort. For example, the GOF *Strategy* pattern maps directly to a delegate so that there is no need to create a class hierarchy.
The .NET Framework uses many software patterns. Knowing which ones that are being used helps in our understanding of the various libraries in the Framework.

18.12 Exercises and Projects