Chapter System Configuration and Creational Patterns in .NET

1. Introduction and Objectives

In this chapter we continue our discussion of delegates that we introduced in chapter XXX. In that chapter we introduced the main topics accompanied by examples and we also discussed the use of delegates in the .NET framework. We introduced the foundation upon which we build more complex designs. In particular, we discuss a number of standard creational patterns in .NET and we also show how to implement them using delegates. We compare the resulting code with that produced using the object-oriented approach in GOF 1995 and POSA 1996. We use modelling techniques such as aggregation and composition (and more generally the *Whole-Part* pattern) together with C# generics, the *Tuple* type and lambda functions to reengineer these design and system patterns.

In general, the topics in this chapter should be easy to understand because delegates are similar to generic function wrappers in C++ 11 and Boost, for example *std::function*, *boost::function* and *boost::signals*. One can even argue that the approaches taken in C++ and C# are really the same, give or take some small syntactical differences between the two languages. Finally, we review how delegates realise *policy-based design* that we discussed in previous chapters. Much of the material in this chapter is also of direct applicability to C++. In this sense there is almost a one-to-one correspondence between the two languages.

This chapter introduces several new design techniques that represent a complete reengineering of the GOF patterns. The approach allows us to choose the most appropriate creational pattern for a given problem:

- It can be applied to any UML class diagram or component diagram of any complexity. This is in contrast to GOF creational patterns that do not scale well to complex object networks.
- The approach subsumes the patterns in GOF 1995 and POSA 1996 as special cases. It is top-down and repeatable rather than bottom-up and idiosyncratic.
- Recent language features such as tuples, delegates and interfaces simplify the process of implementing design patterns in languages such as C++ and C#.
- In some cases it even becomes unnecessary to determine which GOF pattern to use because our process does not depend on the ad-hoc methods that are used for pattern discovery.

We discuss these topics in detail in this chapter.

2. An Analysis of GOF Creational Patterns

In this section we give an account of the common characteristics that the GOF creational patterns share. We pay particular attention to the UML object structures that are used to describe the patterns and the interfaces of factory objects. Having done that we will be in a position to find the limitations of the patterns.

The GOF design patterns are at least twenty years old and they have not been revised to any appreciable degree it seems (at least, not formally) after their appearance in GOF 1995. This also holds for the GOF creational patterns. First, we give some personal opinions concerning these creational design patterns:

a) There seems to be a lack of standardisation on how to implement and use the patterns.
b) It is not clear how to scale the patterns to large object networks. In particular, no discussion is given on how to configure object networks from database tables, XML files and dynamic link libraries (DLLs), for example. We would also like to serialise the object network structure.
c) It is not clear whether the GOF creational patterns encompass all cases of interest to developers. For example, we would like to extend the *Builder* pattern to larger and more complex structures than those described in GOF 1995.
d) There is a possibility (depending on the application) that many factory classes will need to be created and maintained. Some of these classes may have few methods and possibly even no member data. This can lead to maintenance issues.
e) Looking back, it is unfortunate that the *Singleton* pattern is not an easy pattern to implement correctly. Many discussions in articles and books have addressed the problem of implementing this pattern but it would seem that no best solution has been found to date. To compound issues, lifecycle management of thread-safe singletons is difficult to realise. For these reasons and the fact that the added value of this pattern is marginal in our opinion we have decided not to discuss it in this book.

Our aim in this chapter is to address these problems by redesigning creational patterns and using some of the new language features that we discussed in the previous chapter. We focus on the following patterns:

- *Factory Method*
- *Abstract Factory*
- *Builder*

We shall also see how the *Prototype* pattern is directly supported in .NET.

2.1 An Example

Before we discuss multi-paradigm creational patterns we first consider a simple case of a class `Point` that models two-dimensional points. We compute the distance between two points. The corresponding algorithm should be customisable and interchangeable at run time. This goal cannot be achieved in one design iteration and for this reason we propose a number of solutions, each one being slightly more flexible than its predecessor:

- Level 1: The code to compute the distance between two points is hard-wired in class `Point`. It is not possible to choose a different algorithm unless we modify and recompile the code. We also note a potential loss of reusability because the algorithm cannot be used independently of `Point`.
- Level 2: Create a class (this could possibly be a static class) and then let `Point` delegate to it. The issue is to decide who creates a suitable instance of this class and how it is uses by `Point`. We can choose between a state-based and a stateless solution as already discussed in this book. In principle this level corresponds to a small improvement to the flexibility that we achieved at Level 1.
- Level 3: We generalise the class at Level 2 by implementing the GOF *Strategy* pattern using interfaces or abstract base classes. Our main concern is to decide how to instantiate the concrete strategy classes, which modules are responsible for their instantiation and where in the object structure this is realised.
- Level 4: The tight coupling between class `Point` and the corresponding strategy classes will now be loosened by employing delegate types rather than interfaces or abstract base classes. We need to discuss how to create a delegate instance and make it known to `Point`.
- Level 5: Using the .NET *Reflection* API to create objects at run-time.

We have already discussed this problem in chapter 2 where we motivated the transition from an object-oriented programming model to a more flexible one in the context of C++. In this section, the focus is on how, when and by whom objects are created in the .NET Framework.
We start with the following C# code to model points in two dimensions:

```
public class Point
{
  // Point class
  private double xc;        // Space for x-coordinate
  private double yc;        // Space for y-coordinate

  // Constructors
  public Point()
  { // Default constructor

     xc=0.0;
     yc = 0.0;
  }

  public Point(Point s)
  { // Copy constructor

     xc = s.xc;
     yc = s.yc;
  }

  public Point(double x, double y)
  { // Constructor with coordinates

     xc = x;
```

```
      yc = y;
   }

   public double X()
   {   // Return the x-coordinate

      return xc;
   }
   public double Y()
   {   // Return the y-coordinate

      return yc;
   }

   public void X(double x)
   {   // Set the x-coordinate

      xc = x;
   }


   public void Y(double y)
   { // Set the y-coordinate

      yc = y;
   }

}
```

In the following sections we shall refer to this class as the *server*.

2.1.1 Level 1 Solution

The server has minimal functionality for creating points and accessing their member data. We now calculate the distance between two points using the Pythagorean algorithm. To this end, we create a new method in `Point`:

```
public double Distance(Point p2)
{ // Distance between 2 points using Pythagoras' formula

  double d1 = xc - p2.xc;
  double d2 = yc - p2.yc;

  return Math.Sqrt(d1*d1 + d2*d2);
}
```

We can now use this new functionality in client code as follows:

```
Point p1 = new Point();            // Create point

p1.X(1.0);                         // Set x-coordinate
p1.Y(1.0);                         // Set y-coordinate

Point p2 = new Point();

// Print points
Console.WriteLine("Point({0}, {1})", p1.X(), p1.Y());
Console.WriteLine("Point({0}, {1})", p2.X(), p2.Y());

double distance = p1.Distance(p2);
Console.WriteLine("Distance is: {0}", distance);
```

This solution is inflexible because server code must be modified and recompiled when a new type of algorithm is needed. A possible issue is whether the server satisfies the *Single Responsibility Principle* (SRP) because class `Point` is responsible for creating points as well as calculating the distance between them. This solution constitutes a very simple creational pattern.

2.1.2 Level 2 Solution

In this case we encapsulate the code for the distance algorithm in a dedicated class. Then the server can delegate to this new class:

```
public class DistanceCalculation
{
  public DistanceCalculation() {} // Stateless algo

  public double Distance(double x1, double y1, double x2, double y2)
  {
     double dx = x1 - x2;
     double dy = y1 - y2;

     return Math.Sqrt(dx * dx + dy * dy);
  }
}
```

We need to modify the code in the server by adding new member data to it:

```
public class Point
{
  // Point class
  private double xc; // Space for x-coordinate
  private double yc; // Space for y-coordinate

  private DistanceCalculation algo;

  // Constructors
  public Point()
  { // Default constructor

     xc=0.0;
     yc = 0.0;
     algo = new DistanceCalculation();
  }


  // Other code …

  public double Distance(Point p2)
  { // Distance between 2 points using Pythagoras' formula

     return algo.Distance(xc, yc, p2.xc, p2.yc);
  }
}
```

This solution is more flexible than code produced at Level 1 because the code that implements the distance algorithm has been migrated to the class `DistanceCalculation`. On the other hand, the server is still tightly coupled to this class. It is thus not possible to use other algorithms with the server unless of course we modify the code in `DistanceCalculation` and then recompile it. But this solution is less than satisfactory because the server is still tied to a single algorithm.
We also see that `DistanceCalculation` is instantiated in the server, which in fact gives the server the status of a mini-factory.

2.1.3 Level 3 Solution

We extend the solution at Level 2 by implementing the GOF *Strategy* pattern using an interface. The server only knows about an interface and not its implementations:

```
public interface ICalculateDistance
{
  // Calculate the distance. We can't pass a point because then we need to know about
  // point but the point must know about this interface. Then we get circular dependencies.

  double Distance(double x1, double y1, double x2, double y2);
}
```

We are initially interested in two specific algorithms that we encapsulate in the following classes:

```
public class PythagorasDistanceCalculation: ICalculateDistance
{
  public double Distance(double x1, double y1, double x2, double y2)
  {
     double dx=x1-x2;
     double dy=y1-y2;

     return Math.Sqrt(dx*dx+dy*dy);
  }
}


public class TaxiDriverDistanceCalculation: ICalculateDistance
{
  public double Distance(double x1, double y1, double x2, double y2)
  {
     double dx=x1-x2;
     double dy=y1-y2;

     return Math.Abs(dx) + Math.Abs(dy);
  }
}
```

The code in the server needs to be modified to make it more generic. Specifically, it delegates to the interface `ICalculateDistance` which will be instantiated elsewhere:

```
public class Point
{ // Point class

  private double xc; // Space for x-coordinate
  private double yc; // Space for y-coordinate
  private ICalculateDistance algo;

  // Constructors
  public Point(ICalculateDistance algorithm)
  { // Default constructor

     xc=0.0;
     yc = 0.0;
     algo = algorithm;
  }

  // other methods etc.

  public double Distance(Point p2)
  { // Distance between 2 points using Pythagoras' formula

     return algo.Distance(xc, yc, p2.xc, p2.yc);
```

```
  }
}
```

Finally, when we wish to create an instance of `Point` we provide an implementation of `ICalculateDistance`. To this end, we create basic factory code in the client:

```
// Simple algorithm factory; choose which factory and create it.
ICalculateDistance algo;

Console.WriteLine("1. Pythagoras, 2. Taximan");
int choice = Convert.ToInt32(Console.ReadLine());

if (1 == choice)
{
  algo = new PythagorasDistanceCalculation();
  algo2 = new TaxiDriverDistanceCalculation();
}
else
{
  algo = new TaxiDriverDistanceCalculation();
  algo2 = new PythagorasDistanceCalculation();
}
```

Having created the algorithm we can then create points and calculate the distance between them:

```
Point p1 = new Point(algo);        // Create point
p1.X(1.0);                         // Set x-coordinate
p1.Y(1.0);                         // Set y-coordinate

Point p2 = new Point(algo2);

double distance = p1.Distance(p2);
Console.WriteLine("Distance p1->p2 is: {0}", distance);
distance = p2.Distance(p1);
Console.WriteLine("Distance p2->p1 is: {0}", distance);
```

We now see that server code is more flexible than the code from previous solutions. We note however, that each instance of `Point` is constructed with a specific algorithm which may lead to possibly unexpected results as the above test shows. For example, the distance function is not symmetric in the sense that the distance from point `p1` to point `p2` is not necessarily the same as the distance between `p2` and `p1`. In that case it may be advisable to create a singleton algorithm object that all points share.

2.1.4 Level 4 Solution

One of the disadvantages of the Level 3 solution is that client code needs to implement `ICalculateDistance`. This may not always be desirable (or even possible), for example we may have existing code that implements the distance algorithm in some way. Since `ICalculateDistance` has only one method we consider replacing it by a delegate type:

```
public delegate double DistanceDelegate (double x1, double y1, double x2, double y2);
```

The server code needs to be modified as follows:

```
public class Point
{
  // Point class
  private double xc; // Space for x-coordinate
  private double yc; // Space for y-coordinate

  private DistanceDelegate algo;

  // Constructors
```

```
  public Point(DistanceDelegate algorithm)
  { // Default constructor

     xc=0.0;
     yc = 0.0;
     algo = algorithm;
  }

  // Other methods etc.

  public double Distance(Point p2)
  { // Distance between 2 points using Pythagoras' formula

     return algo(xc, yc, p2.xc, p2.yc);
  }
}
```

We now have the luxury of defining algorithms in any way that we like as long as we conform to the signature of the delegate type. For example, we have redefined the above algorithm classes by converting them to static classes. In the case of the Pythagoras algorithm this becomes:

```
public static class PythagorasDistanceCalculation
{
  public static double Distance(double x1, double y1, double x2, double y2)
  {
     double dx=x1-x2;
     double dy=y1-y2;

     return Math.Sqrt(dx*dx+dy*dy);
  }
}
```

We have also created two new algorithms as the following static methods:

```
public static class Algorithms
{ // More mathematical formulae; a point seen as a vector

  public static double MaxNorm(double x1, double y1, double x2, double y2)
  {
     double dx = x1 - x2;
     double dy = y1 - y2;

     return Math.Max(Math.Abs(dx), Math.Abs(dy));
  }

  public static double TruncatedNorm (double x1,double y1,double x2,double y2)
  { // To the nearest integer

     int dx = Convert.ToInt32(x1 - x2);
     int dy = Convert.ToInt32(y1 - y2);

     return Math.Max(Math.Abs(dx), Math.Abs(dy));
  }
}
```

An example of use is :

```
DistanceDelegate algoI = Algorithms.TruncatedNorm;
DistanceDelegate algoII = Algorithms.MaxNorm;

Point pA = new Point(0.0, 0.0, algoI);
Point pB = new Point(1.00001, -1.00025, algoII);
```

```
Console.WriteLine("Distance pA->pB is: {0}", pA.Distance(pB));
Console.WriteLine("Distance pB->pA is: {0}", pB.Distance(pA));
```

We see that delegates offer more possibilities than interfaces for this case.

2.1.5 Level 5 Solution

All the solutions until now demand that source be modified or created and recompiled if a new algorithm is needed and used. In particular, the *Main* method needs to be modified but this situation can be avoided by using .NET *Reflection* and assemblies. To this end, we can create an assembly and load it at run-time. In the assembly the code for the algorithms at level 3 are to be found. We can search in the assembly for the first occurrence of the name `ICalculateDistance` and when we have found it we can then instantiate the class that implements it using the `Activator.CreateInsance()` method (see Duffy 2013 for a discussion). Figure 1 shows the relationship between the software components for the current solution.

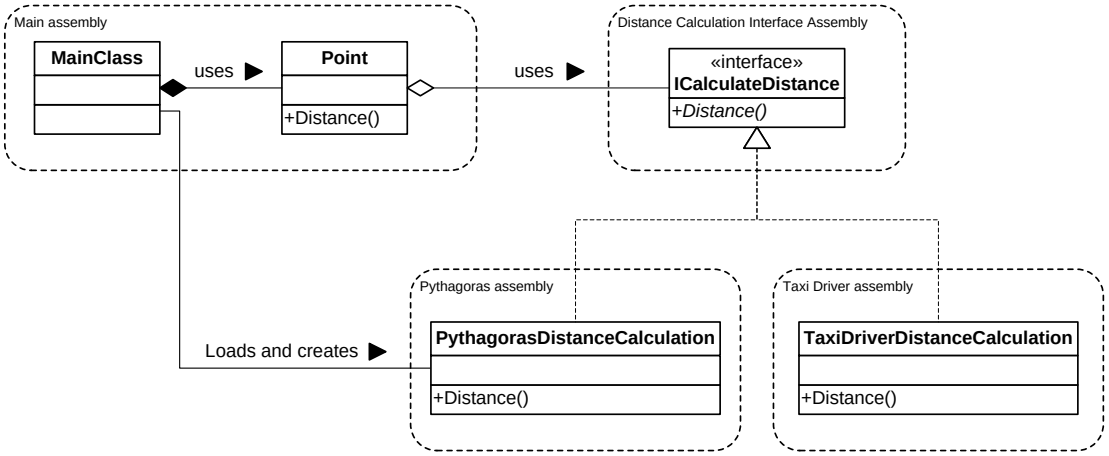

Figure 1 Dynamic Assembly Loading

We shall discuss these issues in more detail in a later chapter.

3. Creational Patterns using the Delegates Approach

We now redesign the GOF creational patterns by using a number of language features in C#, in particular delegates and tuples. A tuple is a collection of related types and .NET supports generic tuples with up to eight generic parameters. The advantage of using tuples is that they model logically related groups of entities as one unit. We use them as input arguments to methods but more importantly in the current context we use them as return types of methods, for example when we create several related objects in an object creational pattern. Then the client can access any of the created objects at will. We give examples of the most important use cases such as creating tuples, accessing the elements of a tuple and using tuples as input arguments to, and as return values from methods. These examples motivate how to use tuples when implementing the new version of the GOF creational design patterns. We take an example of a tuple that models information relating to a person, for example, name, date of birth and ID. First, we can create tuples in a number of ways:

```
static void TestTuple()
{ // Show how to use Tuple

  // Create tuples
  var person1 = new Tuple<string, DateTime, int> ("John", new DateTime(1945,1,1), 123);
  Tuple<string, DateTime, int> person2 = Tuple.Create
                    ("Mary", new DateTime(1949, 12, 12), 567);

  // Access the elements of the tuple
  Console.WriteLine("Elements of tuple {0}, {1}, {2}",
                    person1.Item1, person1.Item2, person1.Item3);
}
```

In the interest of uniformity and standardisation we note that it is possible to create tuples containing one field. An example of use is:

```
static Tuple<T> CreateSingleTuple<T>() where T : new()
{ // Show how to use Tuple with one element

  return new Tuple<T>(new T());
}
```

We now discuss how to use tuples with methods, in particular as return types and input arguments, for example:

```
static Tuple<string, DateTime, int> CreateTuple()
{ // Tuple as return type

  return new Tuple<string, DateTime, int> ("Test", new DateTime(2013, 1, 1), 897);

}
```

and

```
static void PrintTuple(Tuple<string, DateTime, int> tuple)
{ // Tuple as input argument

  Console.WriteLine("Elements of tuple {0}, {1}, {2}",
            tuple.Item1, tuple.Item2, tuple.Item3);
}
```

It is possible to create *nested tuples*. These are tuples whose fields may also be tuples. The rationale is that we can model tree structures whose nodes are tuples or other types. The advantage of this design approach is that can model object structures and networks of arbitrary depth and complexity. We shall see examples of use when we design and implement creational design patterns. Let us take an initial example based on existing code. We create a 'wrapper' method that returns a nested tuple:

```
static Tuple<string, Tuple<string, DateTime, int>, string> CreateNestedTuple()
{ // Nested Tuple as return type

  var person = new Tuple<string, DateTime, int>
("Test", new DateTime(2013, 1, 1), 897);
            return Tuple.Create("<header>", person, "<footer>");

}
```

Finally, we give the code that shows how to use the above method. It should be clear what the code is doing, in particular accessing the elements of a tuple:

```
// Initial tuple examples
TestTuple();
var tuple = CreateTuple();
PrintTuple(tuple);

// We can create tuples with 1 element
Tuple<double> singleTuple = CreateSingleTuple<double>();
//singleTuple.Item1 = 3.14159; // Compiler error; tuples are immutable
Console.WriteLine("Element of single tuple {0}", singleTuple.Item1);

Tuple<DateTime> dateTuple = CreateSingleTuple<DateTime>();
Console.WriteLine("Element of single tuple {0}", dateTuple.Item1);

// Nested tuples and readability
Tuple<string, Tuple<string, DateTime, int>, string> nestedTuple = CreateNestedTuple();
```

```
var person = nestedTuple.Item2;
Console.WriteLine("Internals of nested tuple {0}, {1}, {2}",
person.Item1, person.Item2, person.Item3);
```

The above examples and discussion can be applied and generalised to many application areas. Our interest in this chapter is to use tuples in the context of creational design patterns.

3.1 New-Style Creational Patterns: Motivation and Rationale

A creational pattern creates and initialises one or more logically related objects that client code uses to start and run an application. In the interest of interoperability and maintainability we decide to use delegates to define the interface between the client and factory classes. Regarding the signature of the delegate we model the return type as a tuple having a certain structure (it could be a nested tuple if we initialise more complex object networks, for example) while in general the delegate has no input parameters.

The basic steps are:

1. Determine which objects to create in the object network.
2. Define the factory interface as a delegate type.
3. Create the delegate instances that correspond to concrete factories.
4. Use the concrete factories in client code to configure and run the application.

These are the basic design techniques that we use in the following sections. In this way we can create clean interfaces, standardise the creational design patterns somewhat and in general make them easier to understand and to apply.

3.2 Abstract Factory and Factory Method Patterns

The *Abstract Factory* pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. This is a general description and we make it more precise:

a) We define what is meant by the term *related and dependent objects* (for example, they can form inheritance or composition relationships) and we show how to relate them by using `Tuple<>`. Since this latter class can be nested and can have up to eight generic arguments we see that we can model large and complex object networks.
b) We shall see that the amount of subclassing and the corresponding number of overridable methods is drastically reduced. Instead of having to implement a collection of factory methods (one method per class) we need one interface that we define as a delegate type whose return type is a tuple. The tuple fields contain the desired class instances that will be used to construct the object network.
c) The approach that we introduce here subsumes the GOF creational design patterns as special cases. We thus have a defined process for creating objects of arbitrary depth and complexity, for example, object graphs, composites (GOF 1995) and *Whole-Part* structures (POSA 1996).
d) The resulting software base is easier to maintain than that produced by application of the equivalent object-oriented patterns.

3.2.1 Abstract Factory Patteren

The *Abstract Factory* pattern can be used to create interchangeable product families, in the current context instances of derived classes that are logically related to each other in some way. Some typical examples are:

- Creating multiple class hierarchies of related device-dependent graphical user interface (GUI) widgets, for example buttons, text boxes and scroll bars (Ezust and Ezust 2007). It is obvious that we should configure a toolbox of GUI widgets from a single vendor and not mix widgets from multiple vendors.
- There are a number of requirements when creating integrated hardware/software systems, for example device and vendor-independence and the ability to configure an application for different kinds of software and hardware environments. In this case we use *Abstract Factory* to create a hierarchy of interoperable hardware classes that are modelled using the *Bridge* pattern. These device-dependent classes are precisely the *implementor classes* in this pattern.
- In general, an important requirement is to provide a class library of products by exposing their interfaces and not their implementations.

We now give the '101' example to show how to implement the *Abstract Factory* pattern using both the approach in GOF 1995 as well as the delegate-based approach. We have deliberately chosen the classes that

we instantiate to be as simple as possible. However, these classes are representative of more complex class hierarchies in –production software systems. We shall show how to *morph* this example to more complex cases by incrementally adding more classes and functionality. The UML diagram for a two-product family is shown in Figure 1. In general, we create specific factories that create and associate instances of the derived classes. One of the advantages of the *Abstract Factory* pattern is that it produces families of consistent products.

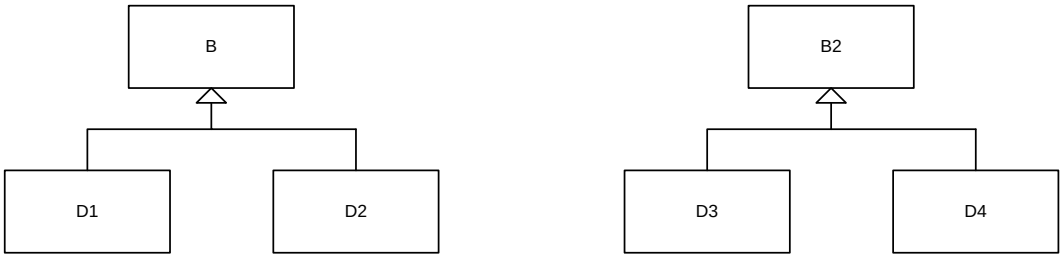

Figure 2 Prototypical Product Hierarchy

The C# code for these classes is:

```
public abstract class B
{ // Base class of product hierarchy

  public abstract void print();
}

public class D1 : B
{
  public D1() { Console.WriteLine("D1 created"); }
  public D1(string s) { Console.WriteLine(s); }

  public override void print() { Console.WriteLine("I am a printed D1");}
}

public class D2 : B
{
  public D2() { Console.WriteLine("D2 created"); }
  public D2(string s) { Console.WriteLine(s); }

  public override void print() { Console.WriteLine("I am a printed D2"); }
}

public abstract class B2
{  // Base class of product hierarchy

  public abstract void print();
}

public class D3 : B2
{
  public D3() { Console.WriteLine("D3 created"); }

  public override void print() { Console.WriteLine("I am a printed D3"); }

}

public class D4 : B2
{
  public D4() { Console.WriteLine("D4 created"); }

  public override void print() { Console.WriteLine("I am a printed D4"); }
```

```
}
```

We now discuss the factories to create instances of these classes. We first discuss the well-known GOF approach. We first create a base factory class:

```
public abstract class GOFAbstractFactory
{
  public abstract B CreateB();
  public abstract B2 CreateB2();
}
```

For convenience, we create a concrete factory class:

```
public class GOFConcreteFactory: GOFAbstractFactory
{
  public override B CreateB() { return new D1(); }
  public override B2 CreateB2() { return new D4(); }
}
```

Next, the following code shows how to use the factory to create objects:

```
Console.WriteLine("** GOF Factory approach **");
GOFAbstractFactory factory = new GOFConcreteFactory();
B bGOF = factory.CreateB();
bGOF.print();
B2 bGOF2 = factory.CreateB2();
bGOF2.print();
```

This code creates two objects that can then be used in an application. We immediately see that this code will need to be modified when the object network contains three objects, for example. Hence we have an issue with maintainability.
We now discuss the same problem by designing factories using delegates. The interface in this case delivers two objects that are encapsulated in a tuple:

```
// Delegate type that returns a pair of objects
public delegate Tuple<T1, T2> Factory<T1,T2>();
```

We immediately see the difference with the GOF approach in which case we need a factory method for each concrete product . In the current case we have a single method that is responsible for the creation of two objects and returning them as a tuple. In fact, the delegate is a specification of a factory method. We can assign this delegate type to the following delegate instances:

```
// Concrete factories for '101' class hierarchy
public static Tuple<B, B2> CreateD1D3()
{
  return new Tuple<B, B2>(new D1(), new D3());
}

public static Tuple<B, B2> CreateD1D4()
{
  return new Tuple<B, B2>(new D1(), new D4());
}
```

We now use these concrete factories to create the family of concrete products using function code operator notation:

```
// Delegate approach to same problem
Console.WriteLine("** Delegate Factory approach **");
Factory<B, B2> factory2 = CreateD1D4;
Tuple<B, B2> result2 = factory2();
```

```
Console.WriteLine("B and B2 derived classes");
result2.Item1.print();
result2.Item2.print();

// Another factory

// Choose factory and run it
Factory<B, B2> factory1 = CreateD1D3;

// Accessing the components of the returned tuple
Tuple<B, B2> result = factory1();
Console.WriteLine("B and B2 derived classes");
result.Item1.print();
result.Item2.print();
```

The generalisation of this pattern to tuples with more than two fields is not difficult. In this way it is possible to configure complex object networks that are described by UML class and component diagrams.

3.2.2 Factory Method (Virtual Constructor) Pattern

The *Factory Method* pattern defines an interface for creating an instance of a single class. Concrete classes that implement the interface are responsible for the actual instantiation of the class. Incidentally, the *Abstract Factory* can be seen as a collection of factory methods as we have seen in the previous section. We take another viewpoint by considering the current pattern as a special case of *Abstract Factory* in which a tuple with one field is created. For example, we can define the delegate type:

```
public delegate Tuple<B> TupleFactory();
```

and assign it to a delegate instance as we did in section 3.2.1. We leave this to the reader. Instead, we define the following non-generic and generic interfaces that correspond to factory interfaces:

```
public delegate B BFactory();
public delegate T Factory<T>();
```

Both of these forms subsume the functionality that the *Factory Method* pattern in GOF 1995 offers. We take an example of use again by examining the classes in Figure 2:

```
// Factory Method
BFactory bf= CreateD1;
B b = bf();
b.print();

bf = CreateD2;
b = bf();
b.print();

// Using generic parameters
Factory<B2> bf2 = CreateD3;
B2 b2 = bf2();

Factory<B> bf3 = CreateD1;
b = bf3();
b.print();
```

where the delegate instances are defined by:

```
public static D1 CreateD1()
{
  return new D1();
}

public static D2 CreateD2()
```

```
{
  return new D2();
}

public static D3 CreateD3()
{
  return new D3();
}
```

Admittedly, these delegate instances have a very simple structure. The classes to be instantiated in these cases have no data members but when classes do have such members we need extra functionality in order to initialise this data. This is normally realised by introducing a so-called *data source object* or *layout manager* that knows how to structure the product.

4. Prototype Pattern

This pattern allows us to create new objects by first creating a prototypical instance and then copying that instance into a new object. This is similar to a copy constructor except that the object that is copied from is determined at run-time and not at compile-time. In GOF 1995 it is assumed that polymorphic copies of instances of derived classes need to be created. C# has a standard interface for this:

```
public interface ICloneable
```

This interface supports cloning, which creates a new instance of a class with the same values as an existing instance. It has a single method called `Clone()`. We see that the cloned object is a *deep copy* of the original object. Let us take an example of a class hierarchy in which cloning has been implemented. A good design principle is to define an abstract base class that abstractly implements the interface while derived classes must implement the interface. We also note that each derived class must have a copy constructor. To this end, the code for a small hierarchy of two-dimensional shapes is:

```
public abstract class Shape: ICloneable
{ // Base class, traditional cloning

  public Shape() { }
  public abstract object Clone();
  public abstract void Print();
}

public class Point : Shape
{ // Minimal class

// For convenience
public double x;
public double y;

public Point() { x = y = 0.0; }
public Point(Point p2) { x = p2.x; y = p2.y; }

public override object Clone() { return new Point(this); }
public override void Print()
{
  Console.WriteLine("I am a printed Point {0}, {1}", x, y);}
}


public class Circle : Shape
{// Minimal class

  public Circle() { }
  public Circle(Circle c2) { }

  public override object Clone() { return new Circle(this); }
```

```
  public override void Print() { Console.WriteLine("I am a printed Circle"); }
}
```

An example of use is:

```
// Use Prototype (+ ICloneable)
Point p1 = new Point();
Point p2 = (Point) p1.Clone();

// Polymorphic copy
Shape prototype = new Circle();
Shape clone = (Shape) prototype.Clone();
clone.Print();
```

We can check that the clone and the original object are independent (as they should be) because this pattern creates deep copies of objects.
It is possible to create a *prototype manager* that manages a registry of available prototypes. The type and number of prototypes is dynamic and they are stored and retrieved using a key (we can implement the corresponding data structure as an *associative array* such as a dictionary). We create an initial version of this generic class whose responsibility is to create prototypical objects. We now have a central *clearing house* as it were. The class is generic and we have defined the *generic constraint* stating that the generic underlying type must implement the `ICloneable` interface.

```
public class PrototypeCreator<T> where T : ICloneable
{ // Simple Prototype manager; for dynamic loading, for example

  private dynamic prototype;

  public PrototypeCreator(T original) { prototype = original; }

  public T Clone() { return prototype.Clone(); }
}
```

An example of use is:

```
// Generic types
Point Origin = new Point();        // Prototypical object
Origin.x = 0.0; Origin.y = 0.0;
PrototypeCreator<Shape> prototypeFactory = new PrototypeCreator<Shape>(Origin);
Point p3 = (Point) prototypeFactory.Clone();
Point p4 = (Point)prototypeFactory.Clone();
p3.Print();
p4.Print();

p3.x = -1.0; p3.y = -2.0;
Origin.Print();     // 0,0
p3.Print();         // -1,-2
p4.Print();         // 0,0
```

The advantage in this case is that the prototypical object only needs to be created once and it can be accessed and cloned through the prototype manager.

The discussion concerning *Prototype* thus far is standard in the sense that it is discussed in GOF 1995 (using base classes) and it is directly supported in .NET. One remark that applies to both of these solution needs to be made, namely that the `Clone()` method returns either a reference to a base class or to an `object`. Client code must cast the reference to the appropriate type which is error-prone and can lead to inefficient code. An alternative approach is to employ user-defined or .NET-based generic delegates to define a prototype signature:

```
// Delegate type: this is similar to .NET ICloneable
public delegate T PrototypeClone<T>();
```

We now discuss how this delegate can be used as an alternative to the traditional pattern. To this end, we create a class `C` with a clone method that returns a deep copy of the current object:

```
public C Clone() { return new C(this);}
```

The advantage of this approach is that casting is not needed in client code, for example:

```
// Two delegates for specific classes (non-polymorphic)
PrototypeClone<C> factory;
C cA= new C(42);
factory = cA.Clone;
C cB = factory();
```

For completeness, the class `C` is defined as:

```
public class C
{
  private int value;

  // Constructors
  public C()
  { // Default constructor

     value = 0;
  }

  public C(int value)
  {
     this.value = value;
  }

  public C(C c2)
  {
     value = c2.value;
  }

  public C Clone() { return new C(this);}

  public void Print() { Console.WriteLine(value); }
}
```

Continuing, let us consider for didactic reasons a class `C2` that has exactly the same structure as `C`. Unfortunately, the previous delegate-based factory cannot be used to create instances of `C2` because it is type-specific. Instead, we must create a new factory object as the following sample code shows:

```
PrototypeClone<C2> factory2;
C2 cC = new C2(99);
factory2 = cC.Clone;
C2 cD = factory2();
```

In other words, the generic delegate approach does not allow us to create one factory object that can be used for arbitrary classes. On the positive side, no casting is needed when using the factory but the question is whether this solution has added value in general. However, we can use specific factories to create a *prototype manager* as already discussed in this chapter. We can use any data type in the manager but we need to know what the original data type was:

```
// Using dynamic type instead of generic argument
PrototypeClone<dynamic> factory3;
```

Then this factory can be used with any data type, for example where use a .NET dictionary:

```
// Build up a small repository
SortedDictionary<int, dynamic> repository = new SortedDictionary<int,dynamic>();
repository[0] = factory(); // Instance of C
repository[1] = factory2();// Instance of C2
```

This approach does have its advantages, for example we are able to assemble a number of prototypical objects in a single datastructure to function as a *palette* from which the application user can select an object.

5. Builder Pattern

This pattern seems to be one of the most difficult patterns to apply. *Builder* is used to construct complex object structures based on a representation of these structures. The actual construction process and the representation are uncoupled, allowing the same construction process to be applied to different representations. We dissect this design pattern into a number of attention areas:

a) Modelling complex object structures in UML and implementing them in languages such as C++ and C#. These structures can be instances of the *Whole-Part* pattern (see POSA 1996) and other object structures involving inheritance, aggregation, composition and association relationships. Classes also have data members and discovering these is an integral part of the design.

b) The class model in part a) needs to be instantiated, for example the multiplicity of aggregation relationships, which concrete (derived) classes to use and the instantiation of data members. These planning duties will be the responsibility of a so-called *director* entity.

c) Using the information from steps a) and b) the *builder* entity creates the parts of the object structure (this is called the *product*) and add them to this structure.

d) The client communicates with both the director and builder. The builder creates the product on behalf of the client.

In short, we must know what has to be done to what and in which order! Sequence diagrams help to describe how messages are sent between collaborating objects (as in GOF 1995, for example). We take an alternative view by using a *concept map* as shown in Figure 3 to describe the relationships between the classes and then describing message passing in textual format. The steps that need to be taken in order to create the product are:

1) Choose which concrete builder that will create the product.
2) Choose a director and introduce it to the builder. It will be the responsibility of the director to notify the builder which objects to build and in which order.
3) Commence construction of the product. The builder executes requests received from the director.
4) Clients can access the product once all parts have been created by the builder.

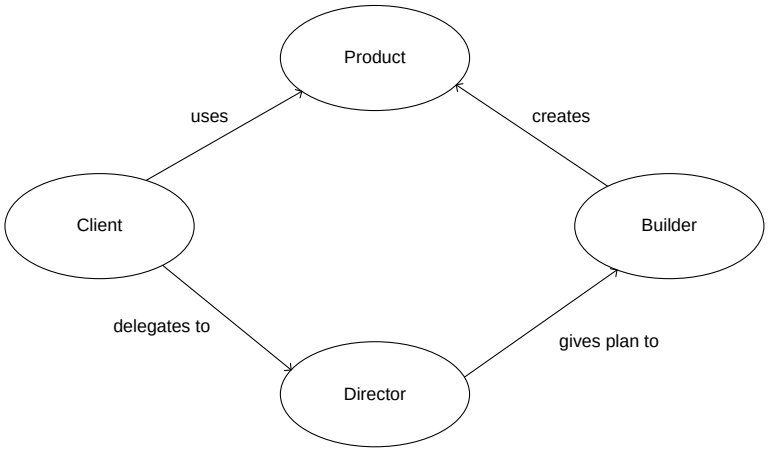

Figure 3 Relationships in Builder pattern

As an example we apply the *Builder* pattern to the construction of the whole-part structure representing a house in Figure 4. This complex object is similar (and slightly simpler than) to the test case in GOF 1995. The classes (deliberately) have minimal functionality at the moment because we wish to focus on essential structure and how to apply the *Builder* pattern to creating the objects in Figure 4. We first describe this object structure in a top-down fashion.

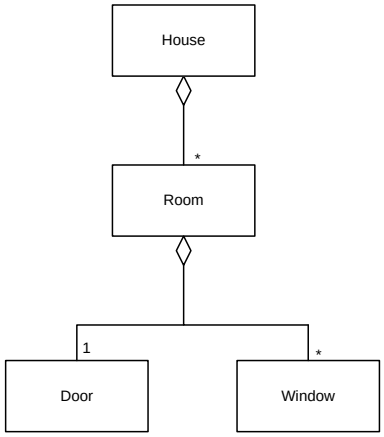

Figure 4 Complex object

We use the .NET class `List<>` to model aggregate objects. The classes are:

```
public class House
{
  // Public members for convenience
  public List<Room> rooms;          // Each house has several rooms

  public House()
  {
     rooms = new List<Room>();
  }

  public void AddRoom(Room room)
  {
     rooms.Add(room);
  }

  public void Print()
  {
     Console.WriteLine("\n** House structure **");
     // Print all the rooms in the house
     for (int n = 0; n < rooms.Count; n++)
     {
             rooms[n].Print();
     }
  }

}


public class Room
{
  // Public members for convenience
  public Door door;                 // Each room has one door
  public List<Window> windows;      // Each room has several windows

  public string nam;
```

```
  public Room(string name, Door door)
  {
     this.door = door;
     windows = new List<Window>();
     nam = name;
  }

  public void AddWindow(Window window)
  {
     windows.Add(window);
  }

  public void AddDoor(Door door)
  {
     this.door = door;
  }
  public void Print()
  {
     Console.WriteLine("\n\t** Room structure ** " + nam);
     door.Print();

     for (int n = 0; n < windows.Count; n++)
     {
             windows[n].Print();
     }
  }

}


public class Door
{
  private int num;

  public Door(int number) { num = number; }

  public void Print()
  {
     Console.WriteLine("\t\t {0}", ToString() + num);
  }
}


public class Window
{
  private int num;

  public Window(int number) { num = number; }

  public void Print()
  {
     Console.WriteLine("\t\t {0}", ToString() + num);
  }
}
```

We see that in order to build this whole-part structure we must first create the parts and then add them for a room and a house to the whole objects to which they belong. To this end, we use `Tuple<>` to model the two whole-part objects:

```
Tuple< Room, Door, List<Window>>
Tuple<House,  List<Room>>
```

This is the design decision that we take in this case. No doubt there are other possible designs and which choice is best becomes clear when you actually start modifying and improving an initial design. In the current case we create a room having one door and two windows and a house consisting of two rooms. We can generalise this design later if necessary. To this end, the factory method to create a room is:

```
static Tuple<Room, Door, List<Window>> CreateRoomTuple(string roomName)
{ // Show how to use Tuple (simple factory)

  // Create objects; hard-coded in this version
  int doorNumber = 1;
  int windowNumber1 = 1;
  int windowNumber2 = 2;

  Door door = new Door(doorNumber);
  Window frontWindow = new Window(windowNumber1);
  Window rearWindow = new Window(windowNumber2);
  Room room = new Room(roomName, door);
  room.AddWindow(frontWindow);
  room.AddWindow(rearWindow);

  return new Tuple<Room, Door, List<Window>>(room, room.door, room.windows);
}
```

Continuing the factory method to create a house consists of two calls to the factory method for rooms:

```
static Tuple<House,  List<Room>> CreateHouseTuple()
{ // Nested Tuple as return type (simple factory)

  // Data needed
  Tuple<Room, Door, List<Window>> roomTuple1 = CreateRoomTuple("Living room");
  Tuple<Room, Door, List<Window>> roomTuple2 = CreateRoomTuple("Dining room");


  House house = new House();
  house.AddRoom(roomTuple1.Item1);
  house.AddRoom(roomTuple2.Item1);

  return new Tuple<House, List<Room>>(house, house.rooms);
}
```

We now give some examples based on this code. First, we create two rooms using the corresponding factory method:

```
// Create rooms
var room = CreateRoomTuple("Living Room");
var room2 = CreateRoomTuple("Dining Room");
room.Item1.Print();
room2.Item1.Print();
```

Next, we create a house using the corresponding factory method:

```
// Create house using a factory
var houseTuple = CreateHouseTuple();
houseTuple.Item1.Print();
```

Finally, we can create a house by creating a number of rooms using the factory method and then adding then to an instance of `House`:

```
// Another way to create a house
House myHouse = new House();

myHouse.AddRoom(room.Item1);
```

```
myHouse.AddRoom(room2.Item1);

// Need another room
var room3 = CreateRoomTuple("Bedroom");
myHouse.AddRoom(room3.Item1);

myHouse.Print();
```

Finally, using delegates to define factory interfaces leads to flexible code:

```
public delegate Tuple<House, List<Room>> HouseInterface();


// Using delegates
HouseInterface houseInterface = CreateHouseTuple;
var houseTuple2 = houseInterface();
houseTuple2.Item1.Print();
```

We are finished. We recommend that you run the code and convince yourself why it gives the following output:

```
  ** Room structure ** Living Room
     Door1
     Window1
     Window2

  ** Room structure ** Dining Room
     Door1
     Window1
     Window2

** House structure **

  ** Room structure ** Living room
     Door1
     Window1
     Window2

  ** Room structure ** Dining room
     Door1
     Window1
     Window2

** House structure **

  ** Room structure ** Living Room
     Door1
     Window1
     Window2

  ** Room structure ** Dining Room
     Door1
     Window1
     Window2

  ** Room structure ** Bedroom
     Door1
     Window1
     Window2

** House structure **

  ** Room structure ** Living room
```

```
    Door1
    Window1
    Window2

** Room structure ** Dining room
    Door1
    Window1
    Window2
```

There are some similarities between the *Abstract Factory* and *Builder* patterns. They both are used to construct complex objects but in contrast to the *Abstract Factory* pattern – where the emphasis is more on creating families of products – the *Builder* pattern returns the product as the final step of a process. The *Abstract Factory* returns the product immediately. Another point to note is that the *Builder* pattern receives its requests from the director which is not necessarily the same as the client. In the case of *Abstract Factory* the burden is on the client (in many cases this could be the `Main()` method) to manage the created objects.

We could think about extending the GOF creational design patterns by combining them in different ways, for example:

- Factory objects that create builder objects.
- Builders that outsource some steps of the construction process to 'subcontractor' factories.

We could think of even more use cases in addition to these but their use may make the code more difficult to understand compared to a more pragmatic (albeit it at the cost of flexibility) solution. In such cases there is only one way to determine which solution is best and that is by actually implementing the pattern and then examining the consequences.
In general, encapsulating object configuration and initialisation code in dedicated classes reduces code clutter and makes the code more readable and maintainable.

6. The Discovery and Design of Configuration and Design Patterns

In this section we formalise the process of how to design and implement creational patterns for arbitrary object networks. The process is repeatable and it describes a step-by-step procedure to determine what the best creational pattern is for a given structural configuration. A precondition to the successful application is that we have already discovered the objects in the application, including object member data, object structure and inter-object relations.

We can categorise objects in terms of their structural complexity as follows:

- C1: 'Simple' objects: there are low-level objects without children and they consist of methods and member data. The *Factory Method* pattern can be used to create these kinds of objects but we also need to extend this pattern to take into account that member data can be initialised from different sources.
- C2: Objects that are composed from other objects. In this case we are referring to aggregation and composition relationships. The *Builder* pattern is useful in this context.
- C3: Object networks that typically use the objects from categories C1 and C2 to form (larger) UML association relationships. In general, the *Abstract Factory* and *Builder* patterns are useful in this context. UML class and component diagrams are recommended as we use them as design blueprints. They improve the readability of the code that implements the modules and interfaces in these diagrams.

We note that creational design patterns are mainly used in relation to objects associated with structural and behavioural patterns but we note that they can also be used with other creational patterns. For example, we can use a factory method to create a builder because we may wish to create application objects in different ways. We can also use a builder (that plays the role of *main contractor*) to create several other builders (the *sub-contractors*), each of which is responsible for the creation of one logical part of a complex object network. In general, creational patterns create objects without having to be concerned with their origins.

We now describe the process to discover creational patterns:

1. Apply the *Builder* pattern to configure the interfaces in the component diagram corresponding to the system's context diagram.
2. Model and design the data related to the interfaces and classes involved in step 1.

3. (the output from steps 1 and 2 is a tuple of application objects).
4. We analyse the top-level application objects in step 3; these could be object networks in their own right but in many cases they belong to categories C1 and C2.
5. Determine which GOF patterns and/or delegate-based patterns to apply to the objects in step 4. A possible design strategy in this step is not to apply design patterns just yet, for example if are developing a prototype (proof-of-concept) application in order to get some insights into the requirements, in which case applying design patterns at this level would be seen as premature optimisation.

We shall see examples and applications of this process in the following chapters.

7. Summary and Conclusions

In this chapter we examined the GOF creational patterns from the viewpoint of the .NET delegate mechanism. Instead of creating special classes and class hierarchies as we see with GOF creational patterns we define factory interfaces using delegate types. Delegate instances are the equivalent of the derived factory classes in the GOF world (the motivation for this approach was the Boost *Functional Factory* C++ library). We compared the GOF approach with the delegate-based approach and we concluded that the latter approach resulted in flexible code and reduced the explosion in the number of factory classes needed to configure an application. Furthermore, we can view all GOF creational patterns as special cases of general *configuration patterns* that produce a tuple of low-level objects or high-level system components that are then used to configure an application.

In order to motivate the use of delegates, we examined a number of examples. First, we took the simple case of a class that has an embedded algorithm. We began with a hard-coded version and we improved the flexibility in a series of iterations, including the GOF *Factory Method* Pattern and a solution based on delegates. We also showed how to create whole-part objects (as discussed in POSA 1996) using a new-style *Builder* pattern based on delegates and tuples.
We shall discuss configuration and creational patterns for large systems (for example, instances of domain architectures) in the next chapter.

8. Exercises and Projects

1. (General Questions)

The objective of this exercise is to compare the GOF creational design patterns with creational patterns based on delegates. Imagine that you have some experience with GOF patterns for medium-sized software projects and that you have read this chapter. Answer the following questions by comparing the two approaches based on the following general criteria:

a) The ability to interoperate with different kinds of software modules (for example, static and normal classes, delegates and interfaces).
b) The ability to extend the functionality as non-intrusively as possible (for example, creating and deploying new factory classes or creating a new application class and corresponding factory classes).
c) How easy is it to understand/learn a subsystem that implements creational patterns for a given application?
d) The amount of effort needed to analyse and debug code for creational patterns?
e) How much effort is needed when we change the code for creational patterns and how stable does it remain after these changes have been brought about?

2. (Software Portability Requirements)

This exercise is a continuation of exercise 1. We define *portability* as a set of attributes that bear on the ability of software to be transferred from one environment to another. We now wish to compare the two creational approaches again with respect to the following subcharacteristics:

- *Adaptability*: the ability to adapt the software to different specified environments without applying other actions (for example, the ability to choose the kind of builder from an assembly (dll) or from a hard-coded default builder that is precompiled into the application).
- *Installability*: this refers to the effort that is needed to install software in a specified environment (for example, how to configure an application from a collection of .NET assembly files).

- *Replaceability*: the opportunity and effort that is needed to replace a given software module by another software module at compile-time or run-time (for example, replacing one set of algorithms by another set of algorithms).

Answer the following questions:

a) In how far can the functionality in the .NET framework be used to allow developers create portable software systems? In particular, focus on techniques such as delegates, interfaces, generics and assemblies.
b) We take another look at the 101 mini-application in section 2.1. How would you now propose a solution that will allow us to satisfy the following requirements?
   i) Configuring the application to support different kinds of algorithms at startup time.
   ii) Loading a new algorithm into the application at run time.
   iii) Replacing an in-memory algorithm by another algorithm at run-time.
   iv) Code generation for an algorithm and using it in the application at run time. Knowledge of .NET *Reflection* and assemblies is assumed.
c) This is a general question on system configuration. In principle we wish to make an application as portable as possible by having the ability to configure each kind of object in the application at both startup time and run time. A typical scenario involves determining which specific builder to choose as well as appropriate sub-builders. Analyse this problem and devise a high-level architecture for the problem.

3. (The *Prototype* Pattern)

In section 4 we discussed the *Prototype* pattern from a number of viewpoints:

a) The GOF approach (in general, using base classes and polymorphic methods)
b) The .NET approach using interfaces (in particular, the interface to create deep copies of objects).
c) Using generic delegates.

We now wish to compare these three solutions to determine which one is *best* (a fuzzy concept) in a particular context. In particular, we focus on issues such as efficiency, standardisation, functionality and maintainability. Compare the solutions a), b) and c) with regard to these four quality characteristics.

4. (Reengineering the GOF Builder Example, small Project)

The creational patterns in GOF 1995 examine a number of examples and test cases, one of which is concerned with modelling a maze for a computer game. Figure 5 depicts the UML class diagram:

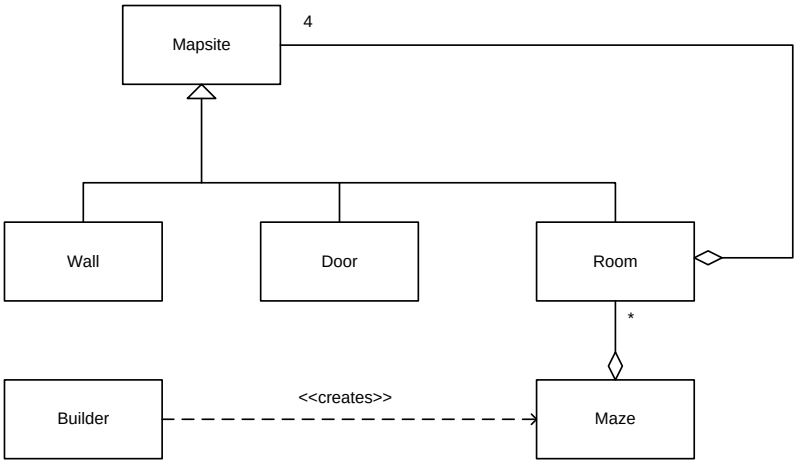

Figure 5 Maze Class Hierarchy

Answer the following questions:

a) Implement the classes in Figure 5 in the C# languge.
b) Create several kinds of builders based on the approach in section 5 of this book.
c) Compare this solution with the solution in GOF 1995. Use the interface specifications from that book to implement your solution.