Chapter An Introduction to .NET Delegates

1. Introduction and Objectives

In this chapter we introduce *Delegates* that are part of the Microsoft .NET. Framework. From a design pattern viewpoint delegates are a realisation of the *Proxy* pattern; in this case a delegate plays the role of the proxy and it has a reference to a *target method* having the same signature as that of the proxy. Clients communicate with the delegate without knowing (or having to know) the details of the target method corresponding to the delegate. It is possible to replace one target method by another one at run-time. In general client code communicates directly with the delegate and not with any target method. A UML diagram showing the structural relationships between these entities is given in Figure 1.

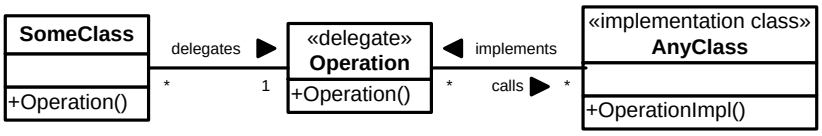

Figure 1 Delegate as proxy (intermediary)

The main goal of this chapter is to define the concepts of *delegate type* and *delegate instance*, how to create and use them in code and how to combine them with other .NET functionality such as generics and lambda functions, for example. We also show how to use delegates to implement the *Strategy* pattern (similar to *plug-in methods*) and the *Observer* pattern using multicast delegates.

The topics that we discuss in this chapter are the building blocks to implement design patterns based on the delegate mechanism. In succeeding chapters we shall discuss how to apply delegates to event models, design patterns and other applications.

We discuss the following topics in this chapter:

- Delegates.
- Delegates as proxies; policy-based design (PBD).
- Plug-in methods.
- Multicast delegates.
- Generic delegate types.
- Action and Function delegates.
- The .NET Standard Event Pattern.
- Delegates with lambda functions.

2. What are Delegates?

A *delegate type* is a specification of a *function signature*, that is it has a return type, a name and zero or more input arguments. In this sense the delegate type represents a protocol or contract. We note that a delegate type is an object but it has no body. In a sense it is similar to an abstract method in that it must be instantiated so that clients can use it. To this end, we introduce the concept of *delegate instance* that is a method whose signature conforms to that of the delegate type.

We take our first example. Consider the delegate type that specifies real-valued functions of a real variable:

```
delegate double ComputableFunction(double x);   // The protocol
```

This type can now be assigned to specific delegate instances if these have the same signature as that of the type. For convenience, we define some instances as static methods as follows:

```
class Delegates

// Some delegate instances
public static double Square(double x)
{
  return x * x;
```

```
}

public static double Cube(double x)
{
  return x * x * x;
}

public static double ModifiedExponential(double x)
{
  return x * Math.Exp(x);
}

//

}
```

Clients can then assign the given type to any of these instances, for example:

```
// Basic usage of delegates
ComputableFunction fun = Delegates.Square;
double val= 2.0;
Console.WriteLine("Square of {0} is {1}", val, fun(val));

fun = Delegates.Cube;
Console.WriteLine("Cube of {0} is {1}", val, fun(val));

fun = Delegates.ModifiedExponential;
Console.WriteLine("Mod. exponential of {0} is {1}", val, fun(val));
```

We see that the client code is shielded from specific functions because it uses the variable `fun` which is a delegate type that can be assigned to any conforming target method. Of course, in this case the client is responsible for the creation of delegate instances and assigning the delegate type to them. In a more general setup the client would not create these instances directly but it would delegate their creation to dedicated factory objects as we shall see in the next chapter.

In general, we can say that a delegate is similar to a *callback* and to C *function pointers*. We now formalise and extend this insight.

2.1 Delegates and Proxy Pattern

In its basic form, a delegate type is an intermediary between server code and the unspecified clients that instantiate the type. In other words, the client has no knowledge of these instances. Furthermore, the multiplicity between type and instance is one-to-one. The type is the surrogate between client and server as can be seen in Figure 1. This suggests that we are discussing an example of a proxy which by definition is a "surrogate or placeholder for another object to control access to it" (GOF 1995). Thus letting the delegate type play the role of a proxy results in a more versatile reference to an object than just a simple pointer or method, for example. This adds to the flexibility and customisability of code. Some common applications are:

a) We can assign a delegate type to an instance method or to a static method. Static methods correspond to global functions with no instance data while instance methods allow us to fine-tune the delegate.
b) *Multicasting*: in contrast to pure proxy behaviour (in which a delegate references a single *target method*) it is possible for a delegate to reference a list of target methods. Calling the delegate results in the target methods being called in the order in which they are added to the delegate. This *multicast capability* has far-reaching consequences for event notification patterns (for example, *Observer*).
c) *Plug-in methods*: instead of implementing algorithmic code directly in a class (thus leading to inflexible software) we embed a delegate as a class member or as an argument in a method of a class. Then the delegate can be assigned to a target method. This capability is similar to that delivered by the GOF *Strategy* pattern. We discuss this issue in section 3.
d) Creating expensive objects on demand (this is called a *virtual proxy*): in general, we outsource object creation to dedicated factories that we implement using delegates. The motivation for this approach is from the Boost *Functional Factory* library (see Demming 2012 for an introduction to this library) and it

results in less *boiler-plate code* than is needed by GOF creational patterns. We discuss how to use delegates to implement creational design patterns in the next chapter.

e) More generally, delegates can be used to implement the different kinds of proxies as discussed in GOF 1995 and POSA 1996. These proxies hide implementation details and delegates are very appropriate in these cases. For example, not only does a proxy offer the services of a component but it performs additional pre and postprocessing, for example access-control checking, incrementing and decrementing a counter, synchronising access to components and creating read-only copies of objects.

Based on this list of examples we see that the use of delegates can help us implement design patterns based on inter-component interfacing.

2.2 Using Delegates in Policy-Based Design (PBD)

We have already discussed PBD and its essential characteristics. Of particular relevance are the concepts of *provides and requires interfaces*. Traditional object-oriented models do not support *requires interfaces* and thus it is not possible to design classes that are able to model events, at least not directly. It is however possible to produce workarounds for this problem (for example, using the *Observer* pattern in GOF 1995). Even then, there are many details that are left to the discretion of the developer. This lack of standardisation leads to code that is difficult to maintain. Instead, we are interested in standard solutions and patterns. For example, we have seen that the Boost *signals* library supports events and the same is true for .NET delegates. An event is defined in terms of two components; first, the *broadcaster* contains a delegate as member. Second, a *subscriber* is a target method that corresponds to the delegate. Finally, an *event* is a construct that exposes delegate features in models involving broadcasters and subscribers.

We take an initial example to motivate why events are important and how to implement them in C#. It is a generic case of a class containing changeable data. When this data changes registered subscribers are informed of the changes, stating what the old value was and what the new value is.

The event construct is standard in .NET. In the current example we are interested in sending an object's old and new values to registered subscribers. The delegate in this case is:

```
public delegate void ChangeHandler(double oldValue, double newValue);
```

The broadcaster class is (notice the event declaration):

```
public class Subject
{ // Broadcaster aka Publisher aka Subject

  private double val;

  public Subject(double myValue) { val = myValue; }

  // The event declaration (the callback in fact)
  public event ChangeHandler ch;

  public double Value
  { // Modify the value and notify subscriber(s)
    // We use .NET properties (setters and getters)

     get { return val; }
     set
     {
             if (val == value) return;
             double oldPrice = val;
             val = value;

             if (ch != null)
             { // The change propagation mechanism

                     ch(oldPrice, val);
             }
     }
  }
```

```
}
```

We now instantiate this class and we also define the subscriber (callback function) that is called when the instance's data has changed. The callback function and test code is given by:

```
class Program
{
  public static void MyChange(double oval, double nval)
  { // Callback function

     Console.WriteLine("Old price {0}, new price {1}", oval, nval);
  }

  static void Main()
  {
     // Define broadcaster
     Subject s = new Subject(3.90);

     // Add/register a subscriber aka observer
     s.ch += MyChange;

     // Trigger the event
     s.Value = 40.0;
  }
}
```

When this program is run the old and new values will be printed (3.90 and 40.0, respectively). It is possible to register multiple callback functions using the delegate multicast capability that we shall discuss in section 4. In section 9 of this chapter we discuss and reengineer this problem from the viewpoint of the *Standard Event Pattern* in .NET.

The above example can be generalised and there are many applications, for example:

- In process control applications, changes in the values of sensors and sensor units (a sensor unit consists of sensors) are conveyed to actuators that use the new sensor values to compute an action.
- Company stock price changes that automatically triggers an option pricing formula.
- A user types a value in a GUI text box that triggers a database query.

3. Plugin Methods and the *Strategy* Pattern

One of the goals of design patterns is to create code that can be modified to suit new requirements. There are many corresponding use cases and solutions and in this section we discuss the particular case of being able to customise the body of a method. To this end, we abstract the body of the method away by using a delegate having the signature that is consistent with that of the method. Instead of hard-coding the method (which is inflexible) we use a delegate in server code that can be assigned to a target method in the client. But where do we define the delegate and how is it used in the server? There are two options that we have already discussed in previous chapters:

- *State-based*: the server is constructed with an embedded delegate.
- *Stateless*: the plugin method is called by providing a delegate as input parameter.

We take an example of a simple class that generates arrays of numbers. We transform an array by applying a function to each of its elements. We consider both the state-based and stateless solutions in a single class for convenience. We recall the delegate type:

```
delegate double ComputableFunction(double x);     // The protocol
```

The class in question is:

```
public class ArrayGenerator
{ // Create an array of values based on a customisable mathematical
  // function that is implemented using delegates
```

```
  private ComputableFunction func;        // Embedded Strategy algo (plugin method)

  public ArrayGenerator(int size, ComputableFunction myFunction)
  {
     func = myFunction;
  }


// State-based delegate form
public double[] ComputeArray(double[] array)
{ // Compute new arrays using the plugin method as data member

  double[] result = new double[array.Length];

  for (int j = 0; j < result.Length; j++)
  {
     result[j] = func(array[j]);
  }
  return result;
}

// Stateless delegate form
static public void ComputeArray(ref double[] array, ComputableFunction plugin)
{ // Compute new arrays using a plugin method as input argument

  for (int j = 0; j < array.Length; j++)
  {
     array[j] = plugin(array[j]);
  }
}

static public void Print(double[] array)
{
     Console.WriteLine('\n');
     for (int j = 0; j < array.Length; j++)
  {
     Console.Write("{0},", array[j]);
  }
}
```

This class is a simple factory to transform an input array to an output array using state-based and stateless delegates. We take an initial example of an array whose elements are squared resulting in a transformed array. We then take the square root of the elements of the transformed array. The following code uses lambda functions (for readability) to define the transformations:

```
// Basic input array
int M = 2;
double[] array = new double[M]; array[0] = 2.0; array[1] = 2.0;

ComputableFunction SquareRoot = x => Math.Sqrt(x);
ComputableFunction Square = x => x*x;

ArrayGenerator generator = new ArrayGenerator(M, Square);
double[] transformedArray = generator.ComputeArray(array);       // Square elements
ArrayGenerator.Print(transformedArray);                          // 4,4

ArrayGenerator.ComputeArray(ref transformedArray, SquareRoot);   // Square root
ArrayGenerator.Print(transformedArray);                          // 2,2
```

The above code is essentially the *Strategy* pattern using delegates. It is more flexible than *Strategy* because the amount of coupling between server and clients is reduced when compared to the object-oriented approach.

A generalisation of this approach is when the algorithm that implements the method has a more complicated structure than a simple loop or function call. For example, the algorithm may have a well-defined structure consisting of several customisable steps. In this case we implement each step using stateless or state-based delegates and this is precisely the *Template Method* pattern as discussed in GOF 1995.

4. Multicast Delegates

It is possible for a delegate instance to reference a list of target methods. The methods are added to and removed from the delegate instance by using the operators += and −=, respectively. We take the same example as in section 2 of this chapter:

```
// Multicast delegates
ComputableFunction d = null;
d += Delegates.ModifiedExponential;
d += Delegates.Cube;
d += Delegates.Square;
value = 3.0;
Console.WriteLine("Multicast delegate of {0} is {1}", value, d(value)); // 9
```

By default, the return value of this list of target methods is that of the last method in the list (in this case the method `Square` and the return value is 9). The other return values are ignored. We can remove target methods as follows:

```
d -= Delegates.ModifiedExponential;
d -= Delegates.Square;
Console.WriteLine("Multicast delegate of {0} is {1}", value, d(value)); // 27
```

A common and useful application of multicast delegates is when several methods operate on a shared resource or object. In this case the return type of each method is `void` and the input argument is a writeable reference.

5. Delegates versus Interfaces

What's the difference between delegates and interfaces? First, both delegate types and interfaces describe abstract behaviour. However, a delegate type is implemented by a delegate instance corresponding to one or more target methods while an interface has one or more abstract methods that are implemented by a single class. In general terms we can say that an interface with one method is similar to a delegate type. However, an interface does not have multicast capability. We now discuss some specific use cases.

5.1 Loose Coupling and Flexibility

In general, the use of delegates leads to more flexible server code than the use of interfaces in our opinion. This is because a delegate type can be implemented by *any* class that has a target method conforming to the signature of the delegate type. This feature adds considerably to the customisability of server code. In the case of using interfaces the server has a reference to an interface that must be implemented by classes at the client site. This means that clients wishing to use server code must implement the interface.

5.2 Delegates and Composition

In general, a delegate can be associated with one or more target methods but server code sees a single delegate. In this sense it is it not possible to define composite delegates. However, we can emulate composite delegates by user-defined code. Some typical use cases are:

a) We create a class that uses one or more delegates. These delegates may be independent of each other or they may be related in some way. Examples where this use case is needed is when we implement the *Template Method* and *Proxy* patterns.

b) We define a class A that encapsulates a related group of delegates. This is an abstract class in the sense that the delegates in A must be associated with target methods in another class B (or in several classes). This use case is similar to the design philosophy underlying the *Bridge* pattern. In this case class A plays the role of the *Abstraction* and class B plays the role of the *Implementor*.

c) We can employ the design technique in use case b) as an alternative to the definition and implementation of interfaces. The advantages of this approach are that the class containing the delegates

may have member data and non-abstract methods, something which is not possible with interfaces (we recall that interfaces may not have member data nor non-abstract methods).

We note that the application of these scenarios in system design leads to flexible applications. In particular, we shall see how to use them as the building blocks for the realisation of many GOF design patterns.
A possible disadvantage of applying the use cases a), b) and c) is that resulting code can be difficult to maintain precisely for the reason that it is loosely coupled. For example, when reviewing server code all we see are delegates because the target methods are in many cases to be found at the client site (and often in a separate file).

5.3 Delegates and Interfaces as a Basis for Design Patterns

Since delegates and interfaces address similar issues we can use them to implement design patterns. In some cases we may prefer to use interfaces and in other cases delegates would seem to be a more appropriate choice. Of course, we can combine the two approaches, for example creating classes that implement an interface and that contain one or more embedded delegates as data members.

The object-oriented model is well-established as a means of implementing design patterns but the model is not necessarily the most effective solution in all situations. One of the challenges is to design a problem without using subtype polymorphism which is one of the lynchpins of the approach taken in GOF 1995. Delegates are a viable alternative and in some cases a more flexible way to implement design patterns but it demands a shift in thinking in terms of deep class hierarchies and virtual methods to a flatter structure involving a smaller set of classes that contain and/or use delegates. We shall see how this is realised in the next chapter when we implement design patterns using delegates.

We conclude this section with some general remarks on the object-oriented and delegate-based approaches in the context of design patterns:

- The object-oriented approach can lead to a proliferation of classes in a class hierarchy because each new class must either implement an interface or be derived from some base class. This situation can lead to maintenance problems. In particular, patterns that can suffer from an explosion in the number of classes are *Command* and *Abstract Factory*. In later chapters we shall see how to implement these patterns using delegates and in the process reduce the number classes that need to be maintained.
- In general, the delegates-based approach is more flexible as it allows easier customisation at the client site. For example, a delegate type can be assigned to either a static method or an instance method.
- We have the possibility to use lambda expressions instead of delegate instances and this can enhance readability of the code in many cases.
- Some design patterns are easier to program and result in more powerful code when they are implemented using delegates. A spectacular example is the *Observer* pattern.

We discuss these issues in more detail in the next chapter.

6. Generic Delegate Types

Since C# supports generics it will not come as a surprise that delegate types may contain generic type parameters. It is also clear that using generic data types rather than hard-coded types promotes the reusability of code. Typical use cases are:

1) Declare the generic delegate type and its generic parameter types; in particular, consider the data types of the input parameters and of the return type.
2) Create a generic class that uses an instance of the delegate type as member or as argument of one of its methods. It is also possible to create a class that uses a delegate instance in which the delegate type's generic parameters have been instantiated.
3) Client code uses the generic class by instantiating its generic parameters. This instantiation process could be outsourced to factories, for example.

We give an example of these use cases by modifying and generalising the code in section 2. In particular, we create delegate functionality that supports generic data types. The new delegate types are defined as:

```
delegate T ComputableFunction<T>(dynamic x);
delegate T BinaryOperation<T>(dynamic x1, dynamic x2);
```

The first delegate type is the generic version of the delegate type in section 2 while `BinaryOperation<>` is a delegate type for binary operations on numeric types. We see that the `ComputableFunction<>` (delegate type has a generic return type and it also has two input parameters whose type is `dynamic`. A dynamic object binds at run-time. The advantage is that we only need to define a single delegate type that works with all numeric types.

The new and extended generic version of `Delegates` (with some additional methods) from section 2 is given by:

```
class Delegates<T>
{
  // Some delegate instances
  public static T Square(dynamic x)
  {
     return (T)x*x; // explicit cast back to T
  }

  public static T Cube(dynamic x)
  {
     return (T)(x * x * x);
  }

  public static T Multiply(dynamic x1, dynamic x2)
  {
     dynamic result = x1*x2;
     return (T)result;
  }

  // A generic method; scaling factor
  public static T Scale<T2> (dynamic x, T2 scaleFactor)
  {
     x *= scaleFactor;
     return (T)x;
  }

}
```

We see that dynamic types in the above code are converted to a generic type. In general, the run-time overhead is minimal. As an example, we first consider multiplying two complex numbers (which are supported in .NET as a data type):

```
// Generic delegates
BinaryOperation<Complex> binaryOperator = Delegates<Complex>.Multiply;
Complex c1 = new Complex(1.0, 2.0);
Complex c2 = new Complex(2.0, 1.0);
Complex cresult = binaryOperator(c1, c2);
Console.WriteLine("Multiply 2 complex numbers {0}, {1}", cresult.Real, cresult.Imaginary);
```

Another example of use is:

```
// Using generic return type of a generic input argument
ComputableFunction<double> function;
double d = 2.0;

function = Delegates<double>.Square;
Console.WriteLine("Square {0}", function(d));

function = Delegates<double>.Cube;
Console.WriteLine("Cube {0}", function(d));

Complex c3 = new Complex(2.0, 1.0);
```

```
ComputableFunction<Complex> complexFunction = Delegates<Complex>.Square;
Console.WriteLine("Square of a complex number {0}", complexFunction(c3));
```

These examples show the power and flexibility of generic delegate types. Not only can we assign a delegate to any target method at run-time but we can also choose the particular data type to use in a given context.

6.1 Generic Methods

A *generic method* is a static method or instance method that has generic input parameters and/or generic return type that differ from the generic types associated with the class which the method is part of. A standard example is a generic method to swap two instances of the same generic type:

```
// Use of a static generic method of class ClassTest
static void Exchange<T>(ref T a, ref T b)
{
  T temp = a;
  a = b;
  b = temp;
}
```

When we use this method it is not necessary to supply the type argument to the method if the compiler does not get confused but it is good programming practice to supply this argument:

```
// Swap 2 objects (generic method)
Complex cA = new Complex(2.0, 1.0);
Complex cB = new Complex(1.0, 2.0);

ClassTest.Exchange(ref cB, ref cA);              // OK if no ambiguity
Console.WriteLine("Complex cA {0}, {1}", cA.Real, cA.Imaginary);

ClassTest.Exchange<Complex>(ref cB, ref cA);     // Avoids ambiguity
Console.WriteLine("Complex cA {0}, {1}", cA.Real, cA.Imaginary);
```

Another example from class `Delegates<T>` is a generic method to scale a generic type by another generic type:

```
// A generic method; scaling factor
public static T Scale<T2> (dynamic x, T2 scaleFactor)
{
  x *= scaleFactor;
  return (T)x;
}
```

We can then call this method as follows:

```
// Scale by an integer (generic method)
int factor = 2;
d = 8.0;
double result = Delegates<double>.Scale<int>(d, factor);
Console.WriteLine("Scaling of {0} by {1} gives {2} ", d, factor, result);
```

6.2 Action and Func Delegates

The .NET Framework supports a small set of delegate types that are so common that they can be used in many applications without developers having to define them themselves. This promotes the readability of the resulting code and it represents a step in standardisation efforts. These delegate types model two kinds of functions:

- *Action*: methods that have `void` return type and can have up to 16 input arguments.
- *Func*: methods that have a non-`void` return type and can have up to 16 input arguments.

In general, you will probably never need a function with so many input arguments; you should consider a redesign if you do have such functions as it probably indicates a design error. How do we use these delegates?

First, the server creates code that uses these delegate types in some way. Then at the client site we assign the delegate to a compatible target method.

For example, consider the following sequence of code snippets. First, we define a function delegate using a lambda expression:

```
// Using a statement block for lambda expressions
// Define an algorithm
Func<Complex, Complex> algorithm = x =>
{ double omega = 3.14159; return new
        Complex (Math.Cos(omega * x.Real), Math.Sin(omega * x.Imaginary)); };
```

We can use this algorithm by applying it to a complex number:

```
// Apply algorithm to a single complex number
Complex com = new Complex(1.0, 2.0);
Complex newNumber = algorithm(com);
Console.WriteLine("Algorithm applied {0}, {1}",
newNumber.Real, newNumber.Imaginary);
```

We can also apply the algorithm to an array of complex numbers:

```
// Apply algorithm to an array of complex numbers
for (int n = 0; n < carray.Length; n++)
{
    carray[n] = algorithm(carray[n]);
}
```

Finally, we can create a method that use a function delegate:

```
// Using .NET Function and Action delegates
public static void Transform(ref T[] array, Func<T, T> algorithm)
{ // Func<int T, out T>

  for (int n = 0; n < array.Length; n++)
  {
     array[n] = algorithm(array[n]);
  }
}
```

We can now use this method by calling it by providing the above function delegate as argument:

```
int sz = 4;
Complex[] carray = new Complex[sz];

Func<Complex, Complex> algorithm = x =>
    { double omega = 3.14159;
    return new Complex (Math.Cos(omega * x.Real), Math.Sin(omega * x.Imaginary)); };

// Now encapsulate algorithm in a target method
Delegates<Complex>.Transform(ref carray, algorithm);

for (int n = 0; n < carray.Length; n++)
{
  Console.WriteLine("{0}, ", carray[n]);
}
```

We conclude this section with a small example on using dynamic variables instead of a function delegate as input argument:

```
public static void Transform(ref T[] array, dynamic algorithm)
```

```
{ // Func<int T, out T>

  for (int n = 0; n < array.Length; n++)
  {
     array[n] = algorithm(array[n]);
  }
}
```

The output is the same as in the previous example. The use of `dynamic` arguments leads to code that is more flexible than is possible with arguments of type `Func<T, T>`. But the choice depends very much on the application in question and what the requirements are.

7. Delegate Compatibility

We now discuss a feature that has to do with the fact that delegate types are incompatible even if their signatures are the same. For example, consider the following code:

```
delegate void DelegateA();
delegate void DelegateB();

public class SomeClass
{
  static public void Method() {}
};
```

We now try to assign these types to each other but in this case a compiler error will result:

```
// Compatibility issues with delegates
DelegateA dA = SomeClass.Method;          // OK
// DelegateB dB = dA;                     // Compiler error, no implicit conversion
```

In other words, the types are distinct.

We consider two delegate instances to be equal if they have the same target methods while two multicast delegates are equal if they reference the same methods in the same order.
The main compatibility issues in the current context are:

- *Parameter compatibility*: when we call a method we can supply arguments that have more specific types than the formal type of the parameters of the method. Similarly, a delegate type can have more specific parameter types than its target methods. This is called *contravariance*.
- *Return type compatibility* : when we call a method the return type may be more specific that the formal return type of the method. In the same way a delegate target method may return a more specific type than the formal return type of the method. This is called *covariance*.

These issues can arise when methods use *object* as either input arguments or as return type. In general, we try to avoid the use of *object* in code as far as possible. We prefer to use generic types. In this way we avoid subtle errors in code.

8. Instance and Static Method Targets

In section 2 we defined a class `Delegates` containing a number of static methods that play the role of delegate instances. We note that it is also possible to assign a delegate type to an instance method. One of the reasons for doing this is that we can fine-tune the corresponding target method because its body can use a mixture of member data and input arguments, a combination that is not possible with static members. For example, we modify the `Delegates` class by adding a new data member called `factor` and a method called `Scale` that can play the role of a delegate instance:

```
private double factor;

public Delegates(double scaleFactor) { factor = scaleFactor; }

// Instance methods
```

```
public double Scale(double x)
{
  return factor*x;
}
```

We see that different results will be produced when the method is called because the results depend on both the member data and on input arguments. An example of use is:

```
// Using instance methods
ComputableFunction delegateType;
double factor = 3.0;
Delegates myDelegate = new Delegates(factor);
delegateType = myDelegate.Scale;

double value = 5.0;
Console.WriteLine("Scaled value of {0} is {1}", value, delegateType(value));

// Another scaling factor and delegate
double factor2 = -3.0;
Delegates myDelegate2 = new Delegates(factor2);
delegateType = myDelegate2.Scale;
value = 2.0;
Console.WriteLine("Scaled value of {0} is {1}", value, delegateType(value));
```

We see in this case (and in general) that the delegate instance is associated with both a method and the object that the method belongs to.

Finally, it is possible to use the following two properties of the class `System.Delegate` to give the target name and object of a given delegate. In the case of a static method there is obviously no corresponding object:

```
// Inquiring about the instance behind a delegate type
Console.WriteLine(delegateType.Method);                  // Double Scale(Double)
Console.WriteLine(delegateType.Target == myDelegate2);   // True

ComputableFunction function = Delegates.Square;
Console.WriteLine(function.Method);                      // Double Square(Double)
Console.WriteLine(function.Target == myDelegate2);       // False
```

9. .NET Events and the Standard Event Pattern

The .NET Framework has a pattern for defining events. Before we discuss the details we give an overview of some issues involved with the *basis of communication* between modules in a software system. In particular, we classify systems according to whether communication between modules depends upon shared state, events or both. To this end, an *event* is a transfer of information occurring at a discrete time. This classification is (see Shaw and Garlan 1996):

a) *Events*: no shared state; all communication between modules relies on events.
b) *Pure state*: communication is solely by means of shared state; the receiver must repeatedly inspect the state variables in the sender to detect changes.
c) *State with hints*: communication is by means of shared state but the receiver is actively notified of changes by an event mechanism. Polling of the state is not needed. The receiver may choose to ignore events and reconstruct all necessary information from the shared state. In this case the events are hints to the receiver.
d) *State and events*: both shared state and events are used. The events are crucial because they provide information that is not available from monitoring of the state.

The *Standard Event Pattern* is an implementation of cases a), c) and d) above. To this end, the predefined class `System.EventArgs` is a base class that contains information pertaining to an event. Subclasses of `System.EventArgs` expose data as properties or as read-only fields. We also need to define the delegate for the event. The rules are:

- The delegate must have `void` as return type.

- The delegate must have two arguments. The first argument is of type *object* and it represents the sender of the event while the second argument contains the extra information to convey to the receiver.
- The name of the delegate must end in *EventHandler*.

In general, we are interested in the generic delegate `System.EventHandler<>` that satisfies these conditions and whose generic parameter corresponds to the class whose data (or a subset of the data) is conveyed from sender to receiver. An UML class diagram to show the relationships between the entities is shown in Figure 2.

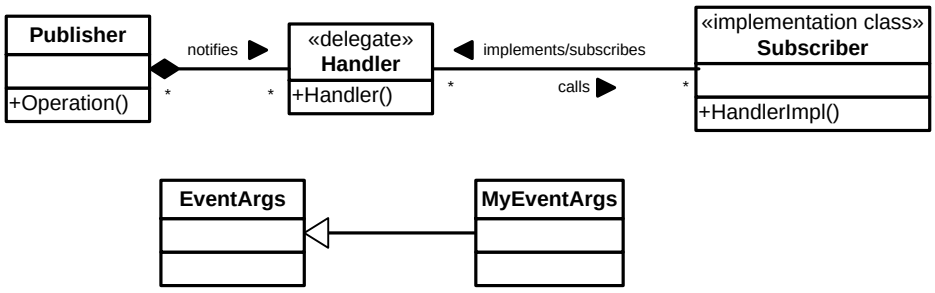

Figure 2 Model for the Standard Event Pattern

We take the example that we introduced in section 2.2 of this chapter and we now implement it using the current pattern. The new code is similar to that in section 2.2 but the solution has now been standardised. The model for the data that is conveyed from sender to receiver is (we retain the code of section 2.2 as comments for comparison purposes):

```
// V1: public delegate void ChangeHandler(double oldValue, double newValue);
public class ValueChangedEventArgs : EventArgs
{ // The class containing the conveyed data from sender to receiver

  public readonly double oldValue;
  public readonly double newValue;

  public ValueChangedEventArgs(double o, double n)
  {
     oldValue = o;
     newValue = n;
  }
}
```

Next, the class that plays the role of broadcaster is:

```
public class Subject
{ // The Broadcaster

  private double val;

  public Subject(double myValue) { val = myValue; }

  // V1: public event ChangeHandler ch;
  public event EventHandler<ValueChangedEventArgs> valueChanged;

  protected virtual void OnValueChanged(ValueChangedEventArgs e)
  {
     if (valueChanged != null) valueChanged(this, e);
  }


  public double Value
  {
     get { return val; }
```

```
        set
        {
                if (val == value) return;

                double oldValue = val;
                val = value;

                // Update the subscriber
                OnValueChanged(new ValueChangedEventArgs(oldValue, val));
        }
    }
}
```

Each time the data in this class changes the subscriber will be updated. For example, a particular case is when there is more than a 10% relative increase in the old value:

```
public static void MyChange(object sender, ValueChangedEventArgs e)
{
  if ((e.newValue - e.oldValue) / e.oldValue > 0.1)
  {
     Console.WriteLine("Alert, more than 10% rise");
  }
     else
  {
     Console.WriteLine("No news, good news");
  }
}
```

Finally, an example of use is:

```
static void Main()
{
  // Define broadcaster with empty event
  Subject s = new Subject(3.90);

  // Define target method for broadcaster's event
  s.valueChanged += MyChange;

  // Change propagation method
  s.Value = 24.0;
}
```

This particular example is interesting in itself because it shows how to apply event notification patterns in C#. The code can be adapted to more general situations such as the *Observer* pattern and applications involving graphical user interfaces (GUIs) and databases.

10. Making Delegates Easier: Lambda Functions

We have seen how to instantiate a delegate type by assigning it to a target method. In general this method is either a static method or instance method of some class. In general, there may be multiple classes in multiple source files containing methods that can play the role of method target for a delegate. It then becomes increasingly difficult to maintain the resulting code base when the number of target methods increases for reasons such as:

- How to remember the names of large number of classes that implement the target methods.
- Remembering what the bodies of the target methods actually do.
- The drudgery of having to create and maintain tiny classes.
- At the client site it is not always clear what the code is doing because the delegate in use is assigned to a target method that is defined somewhere else.

In order to alleviate these problems .NET has the capability to employ *anonymous methods* and *lambda expressions*, the latter subsuming the former as a special case. Thus, we focus on lambda expressions but for historical reasons we give an introduction to anonymous methods. In order to write an anonymous method we

include the *delegate* keyword followed by an optional parameter list and then a method body. We take an example. As before, we define the delegate type:

```
delegate double ComputableFunction(double x);
```

We can now define the following anonymous methods as targets for this delegate:

```
// Anonymous methods
ComputableFunction Log = delegate(double x) { return Math.Log(x);};
ComputableFunction Sin = delegate(double x) { return Math.Sin(x);};
ComputableFunction Cos = delegate(double x) { return Math.Cos(x);};

double v1 = 1.0; double v2 = 3.14159;
Console.WriteLine("Anonymous method: {0},{1},{2} ", Log(v1), Sin(v2), Cos(v2));
```

Now, a lambda expression is an unnamed method that is written in place of a delegate instance. The compiler then converts the lambda expression into either a delegate instance or an expression tree. Examples of lambda expressions are:

```
// Lambda calculus; write Square, Cube and ModifiedExponential
// in lambda form
ComputableFunction SquareII = x => x * x;
ComputableFunction CubeII = x => x * x * x;
ComputableFunction ModifiedExponentialII = x => x * Math.Exp(x);
```

An example of use is:

```
double val = 2.0;
Console.WriteLine("Lambda: {0},{1},{2} ",
SquareII(val), CubeII(val), ModifiedExponentialII(val));
```

We see that we call a lambda function in code just as we would call a 'normal' function.
Concluding, we see that the use of lambda expressions makes code more readable, especially if the corresponding code body is not too long. We no longer use anonymous methods in our applications.

11. Generic Constraints

After having created generic classes and generic methods we can then instantiate them by replacing their generic types by *any* concrete types. Generic classes and methods make certain assumptions concerning the underlying generic type, for example a generic type must be a reference type, a struct or have a default constructor. To formalise this we say that *constraints* can be attached to the generic type (call it T), namely:

- T must be derived from some base class.
- T must implement an interface.
- T is a class (reference-type constraint).
- T is a struct (value-type constraint).
- T must implement a default constructor.

It is possible to have multiple constraints attached to a generic type. (Of course, the reference-type and value-type constraints are mutually incompatible). In order to show how to use these constraints we take the example of a one-dimensional generic interval. The underlying type must be *totally ordered* in the mathematical sense because we wish to be able to compare instances. Thus, we define a number of constraints regarding the type `Range<T>`:

- It must implement a user-defined interface `IPrintable`:

```
public interface IPrintable
{
    void Print();
}
```

- It is a struct.
- It must implement the .NET interface `IComparable<T>` so that we can compare the values of the underlying type.

The class interface is:

```
public class Range<T> : IPrintable
     where T : struct, IComparable<T>
{ // Interval [a, b) (closed-open). T is a totally ordered set (a < b compare)

private T a;
private T b;

private Range() {}

public Range(T left, T right)
{
  if (left.CompareTo(right) < 0)
  {
     a = left;
     b = right;
  }
  else
  {
     b = left;
     a = right;
  }
}

public Range(Range<T> r)
{
  a = r.a;
  b = r.b;
}

public T A // left
{
  get
  {
     return a;
  }
  set
  { // The left boundary value of the range will be set with the value
     a = value;
  }
}

public T B // right
{
  get
  {
     return b;
  }
  set
  { // The right boundary value of the range will be set with the value
     b = value;
  }
}

public T Spread
{
  get
```

```
  {
     dynamic bb = b; dynamic aa = a;
     return (T)(bb - aa);
  }
}

public bool Left(T val)
{
  if (val.CompareTo(a) < 0)
  {
     return true;
  }
  return false;
}

public bool Right(T val)
{
  if (val.CompareTo(b) >= 0)
  {
     return true;
  }
  return false;
}

public bool Contains(T val)
{
  if ((val.CompareTo(a) >= 0) && (val.CompareTo(b) < 0))
  {
     return true;
  }
  return false;
}

public void Print()
{
Console.WriteLine("[{0},{1}), width {2}", a, b, Spread);
}

public T[] Mesh(int nSteps)
  {
     // Need dynamic here as generics.
     dynamic bb = b; dynamic aa = a;
     dynamic h = (bb-aa) / (dynamic)nSteps;
     dynamic val = a;
     T[] result = new T[nSteps + 1];

     for (int i = 0; i < result.Length; i++)
     {
             result[i] = val;
             val += h;
  }

  return result;
  }
}
```

This is a useful class in its own right (for example, it can be used to model intervals that are needed in various kinds of differential equations) and the code shows how to define generic constraints to make it more reliable and to restrict the underlying types that it accepts. We now give some examples of use:

```
Range<double> r1 = new Range<double>(0.0, 1.0);
r1.Print(); // 0,1
```

```
Range<double> r2 = new Range<double>(r1);
r2.Print(); // 0,1
r1.A = 1.0; r1.B = 20.0; // Copy by value

r1.Print(); // 1,20
r2.Print(); // 0,1

// Some tests for set inclusion
double ptL = -2.0; double ptM = 0.5; double ptR = 2.0;
Console.WriteLine("Left: {0}", r2.Left(ptL));
Console.WriteLine("Right: {0}", r2.Right(ptR));
Console.WriteLine("Contains: {0}", r2.Contains(ptM));

// Creating a mesh
int meshSize = 100;
double[] mesh = r2.Mesh(meshSize);

for (int i = 0; i < mesh.Length; i++)
{
  Console.Write("{0 :f2}, ", mesh[i]);
}
```

We now wish to extend `Range<double>` to support a number of methods that are needed for *Interval Arithmetic* which is a branch of Numerical Analysis (see Moore 1966 for an introduction). In particular, we can view intervals as numbers on the one hand and as sets on the other hand. Some methods are to be found in the following code where we define a number of arithmetic and set operations:

```
public class Interval : Range<double>
{ // Interval arithmetic version 0.01

  public Interval() : base(0.0, 1.0) {}
  public Interval(double a, double b) : base(a, b) { }

  // Numerical operations
  public static Interval Add(Interval i1, Interval i2)
  {
     return new Interval(i1.A + i2.A, i1.B + i2.B);
  }

  public static Interval Subtract(Interval i1, Interval i2)
  {
     return new Interval(i1.A - i2.B, i1.B - i2.A);
  }

  public double Width()
  {
     return B - A;
  }

  public double Midpoint()
  {
     return 0.5*(B + A);
  }

  // Set-like operations
  public Interval Union(Interval i2)
  {
     return new Interval(Math.Min(A, i2.A), Math.Max(B, i2.B));
  }

  public Interval Intersection(Interval i2)
  {
```

```
        return new Interval(Math.Max(A, i2.A), Math.Min(B, i2.B));
    }

    public bool Subset(Interval i2)
    {
        return (A >= i2.A && B <= i2.B);
    }

    // Others ...

}
```

Some sample code is:

```
// Interval arithmetic
Interval rA = new Interval(0.0, 1.0);
Interval rB = new Interval(1.0, 2.0);
rA.Print(); rB.Print();

Interval rC = Interval.Add(rA, rB);
Interval rD = Interval.Subtract(rA, rB);

rC.Print(); rD.Print();

// Numeric properties
Interval rE = new Interval(0.0, 1.0);
Console.WriteLine("Midpoint: {0}", rE.Midpoint());
Console.WriteLine("Width: {0}", rE.Width());


// Set properties
Interval rF = new Interval(0.0, 3.0);
Interval rG = new Interval(1.0, 2.0);

Interval rU = rF.Union(rG);
rU.Print();

Interval rI = rF.Intersection(rG);
rI.Print();


Console.WriteLine("Subset?: {0}", rI.Subset(rF));        // True
Console.WriteLine("Subset?: {0}", rI.Subset(rG));        // True

WriteLine("Subset?: {0}", rF.Subset(rI));                // False
Console.WriteLine("Subset?: {0}", rG.Subset(rI));        // True
```

We have created the class `Interval` using inheritance. An alternative would have been to use .NET *extension methods*.

12. Summary and Conclusions

In this chapter we have given an introduction to delegates in the .NET Framework. Delegates are used to model functions and they can be seen as a competitor to interfaces and polymorphic methods in C#. We have shown by discussions and examples how delegates are used to create flexible applications. In particular, we discussed how delegates can be used as the building blocks to help us create next-generation design patterns.

It takes some time to become comfortable with delegates and to apply them in applications. To this end, we describe the steps that need to be taken in a specific scenario, namely creating and instantiating a class that has a delegate as member. The steps are:

1. Define the delegate type that describes the signature of the abstract function.

2. Design a class with a (private) delegate instance as member. This delegate instance should be instantiated in all class constructors. Optionally, a public method to choose another delegate may be needed but this depends on the requirements.
3. Client code instantiates the class and delegate type in step 2. The client then has the responsibility for the creation of methods that have the same signature as the delegate type in step 1 and using as the target methods in the class constructor.
4. Run and test the code.

13. Exercises and Projects

1. (Delegates' Basics)

In this exercise we pose general questions concerning delegates and their relationships with other constructs in C#. Answer the following questions:

a) What are the major differences between a delegate type and an interface? Can an interface contain a delegate type?
b) Compare the flexibility levels in client code when using delegates compared to using interfaces.
c) Consider a class that contains a delegate type as member. What are the advantages of this approach in terms of customisability at the client site?
d) Can we use an interface as data member?

2. (Coupling in Object-Oriented Systems)

Object-oriented software uses a combination of inheritance, composition and aggregation to create networks of (usually) tightly-coupled objects. The objective of this exercise is to consider some typical cases of how these relationships are used, what the consequences are and how to provide alternative approaches to designing software systems. Answer the following questions:

a) We use inheritance for a number of reasons, for example realising subtype polymorphism by means of interfaces and abstract base classes in a class hierarchy. Each time new functionality is needed implies that we create a new derived class that implements the interface's methods. In other words, we can get an explosion in the number of *tiny classes* (that is, classes with very few methods) that we need to create. Instead of creating multiple classes consider creating a single class containing embedded delegates that *emulate polymorphic behaviour*. What are the advantages of this approach and for what class of problems would this approach be useful?
b) The GOF patterns use class hierarchies as described in part a) of the current exercise. In particular, the *Command*, *Abstract Factory* and *Observer* patterns can lead to an explosion in the number of classes that are needed in applications. The issue with *Command* can be particularly acute and it is not uncommon to need many command classes in a typical desktop application.
   How can delegates be used in order to reduce the number of classes in these patterns and hence promote maintainability?
c) The *Bridge* pattern decouples an abstraction from its implementation so that both can vary independently of each other. Determine how to implement this pattern using delegates (hint: you will need to create a class composed of delegates that models the *Implementor* interface in this pattern). Take a concrete example to test your design.

3. (Plug-in Methods)

We create a simple class/struct that models two-dimensional points in Cartesian geometry. The desired functionality is:

a) Constructors (for example, default constructor and constructor with x and y coordinates).
b) Properties to set and get the x and y coordinates of a point.
c) A method to compute the distance between two points using a delegate. First, create a class with static methods for different algorithms to compute the distance between two points. Then design the `Point` class as consisting of a delegate (*state-based*) whose signature is compatible with the signature of the above algorithms. Test your code.
d) Now create a function to compute the distance between two points in which we provide a specific algorithm as input argument (the *stateless* option). Test your code.
e) Compare the stateless and state-based solutions in terms of efficiency, reliability and maintainability.

4. (Standardisation)
Consider the language features that we discussed in this chapter. Which features promote the standardisation of C# code in a team environment by ensuring that the developers in the team adhere to the same standards? For example, one answer would be to use .NET *Function* and *Action* delegates. How would standardisation improve the quality of the code used to implement design patterns?

5. (Delegates and their Suitability for Design Patterns)
In later chapters we shall reengineer the GOF patterns by applying the C# and .NET features that we introduced in this chapter. In particular, the application of generic delegates, interfaces, generic constraints and events will help the developers in their efforts to produce flexible designs. To this end, we have a number of questions to stimulate discussion on how to realise these goals:

a) Give the top five GOF behavioural patterns that could benefit from the application of delegates. Motivate your answers.
b) Why does the GOF *Command* pattern lead to an explosion in the number of tiny classes, each one implementing its own *execute()* method? Consider how you would implement *Command* using just one class containing an embedded delegate as member. What are the advantages and disadvantages of this approach compared to the GOF approach?
c) As a follow-on question from question b), how can the use of delegates reduce the amount of subclassing as seen with GOF patterns? Examine the top three GOF patterns in which class hierarchies form an essential part of their design structure.
d) Compare the GOF *Observer* pattern with the .NET *Standard Event Pattern* in terms of functionality, maintainability and efficiency.
e) How can you re-engineer the GOF *Template Method Pattern* to a design using delegates? Are 'hybrid' (for example, combining interfaces and delegates) solutions possible?

6. We generalise the code for the class `Subject` in section 2.2. First, the underlying data type should be made generic and second the trigger for events should be customisable. Answer the following questions:
   a) Create a class `Subject<T>` that has a generic data type instead of type `double` as in section 2.2.
   b) The precondition for an event to fire, namely:

```
if (val == value) return;
```

   should be replaced by a more generic delegate-based version:

```
public delegate bool ChangePrecondition<T>(T oldValue, T newValue);
```

   c) Test the new code based on the same test program as in section 2.2.
   d) Test the code with various subscribers and event trigger functions.

7. (Event Pattern)
We return to exercise 6. The objective in this case is to reengineer the solution code from that exercise but now using the functionality that the *Standard Event Pattern* delivers (as discussed in section 9 of this chapter). In other words, implement steps a), b), c) and d) of exercise 6 again.
Examine the advantages of using tuples as argument in the event argument class.

8. (Mathematical Operations on Double-Precision Arrays)
The objective of this exercise is to create a number of algorithms that take an array as input. Answer the following questions:

a) Create four static methods to compute the maximum, minimum, sum and average (mean) of the array.
b) Combine and merge the code from part a) to create a tuple having four elements containing the maximum, minimum, sum and average (mean) of the array:

```
public static Tuple<double, double, double, double>
                    Properties(double[] arr)      { // max, min, sum, avg

                                                    // your code here
```

```
}
```

c) Determine in which kinds of applications solution b) is more appropriate than solution a). Consider characteristics such as efficiency, functionality and maintainability.

9. (Emulating C++ STL Functionality)

In this and the next exercise we emulate some of the useful functionality from the C++ Standard Library. This library has several categories of algorithms that operate on *generic containers* as they are called in C++. In order to reduce the scope we examine two versions of an algorithm to count the numbers in a C# collection that satisfy a given condition:

i) Count the number of elements whose values are the same as a given target value (for example, count the number of elements having the value 5).

ii) Count the number of elements satisfying a given condition (for example, the number of even elements or the number of elements whose value is less than 10).

The code for case i) is given for motivation:

```
public delegate bool Condition<T> (T value);

public class CountProperties<T> where T : IComparable<T>
{

  public static int count(IEnumerable<T> coll, T value)
  { // std::count

     int result = 0;

     var en = coll.GetEnumerator();
     T item;

     while (en.MoveNext())
     {
             item = en.Current;

             // Hard-coded test ==
             if (item.CompareTo(value) == 0) result++;
     }

     return result;
  }


}
```

The objective of this exercise is to write the code for part ii) using delegates. In this case the delegate type is:

```
public delegate bool Condition<T> (T value);
```

and the signature of the function to be implemented is:

```
public static int count(IEnumerable<T> coll, Condition<T> condition)
{ // std::count_if

  // TBD
}
```

Test the function using various delegate types as already mentioned in point ii) above.

10. (Emulating C++ STL Functionality)
The objective of this exercise is to emulate the functionality of the numeric algorithm that sums the elements of collaboration in some way:

i) Sum the elements using normal addition.
ii) Sum the elements by providing a delegate as an input argument that operates on all elements of the collection.

Implement both of these algorithms in C#. Base your approach using the code from exercise 9. The specifications are:

```
public delegate T BinaryFunc<T> (T x, T y);

public static double Accumulate(IEnumerable<double> coll, double initValue);

public static double Accumulate(IEnumerable<double> coll, double initValue,
                                BinaryFunc<double> operation);
```

11. (Filtering Algorithms)
In this exercise we emulate some of the *modifying algorithms* in STL:

- Transforming an input collection to an output collection (two forms).
- Replacing values in a collection by a new value (two forms).
- Fill the elements in a collection with a given value.

In the interest of standardisation we define the following functions and predicates:

```
// Functions
public delegate T UnaryFunc<T>(T x);
public delegate T BinaryFunc<T>(T x, T y);

// Predicates
public delegate bool UnaryPredicate<T>(T value);
public delegate bool BinaryPredicate<T>(T value);
```

We also include some specific cases for convenience:

```
public class UnaryPredicates
{

  public static bool IsEven(double x)
  {
     return (x % 2 == 0);
  }

  public static bool IsOdd(double x)
  {
     return (!IsEven(x));
  }

}

public class UnaryFunctions
{

  public static double Square(double x)
  {
     return x * x;
  }

}
```

```
public class BinaryFunctions
{

  public static double Multiply(double x, double y)
  {
     return x * y;
  }

}
```

The objective of this exercise is to implement and test the following algorithms:

```
public class TransformProperties<T> where T : IComparable<T>
{

  public static void Transform(List<T> coll, ref List<T> result, UnaryFunc<T> operation)
  { // std::transform I
    // Copy and modify an input collection to output collection in one step.


  }

  public static void Transform(List<T> coll1, List<T> coll2, ref List<T> result,
                   BinaryFunc<T> operation)
  { // std::transform II
    // Copy and modify 2 input collections to output collection in one step.

    // PRECONDITION: coll1 and coll2 have the same size


  }

  public static void Fill(List<T> coll, T value)
  { // std::fill
    // Assign value to each element of coll


  }


  public static void Replace(List<T> coll, T oldValue, T newValue)
  { // std::replace


  }

  public static void Replace(List<T> coll, UnaryPredicate<T> operation, T newValue)
  { // std::replace


  }

}
```

In order to make it easier how to implement these methods we provide some typical examples of use. In this way we provide you with the input and output specifications. Then the challenge is to connect the input to the output as it were:

```
// Transformation algoritms
// I
List<double> arr1 = new List<double>();
```

```
for (int n = 1; n <= 10; n++)
{
  arr1.Add((double)n);
}

List<double> arr2 = new List<double>();
TransformProperties<double>.Transform(arr1, ref arr2,
                            UnaryFunctions.Square);

// II
int N = 4;
List<double> arrA = new List<double>();
for (int n = 0; n < N; n++)
{
  arrA.Add(1.0 + (double)n);
}

List<double> arrB = new List<double>();
for (int n = 0; n < N; n++)
{
  arrB.Add(10.0 - (double)n);
}

List<double> arrC = new List<double>();
TransformProperties<double>.Transform(arrA, arrB, ref arrC,
                                                  BinaryFunctions.Multiply);

TransformProperties<double>.Fill(arrC, 2.0);
double newValue = 3.0;
TransformProperties<double>.Replace(arrC, 2.0, newValue);

TransformProperties<double>.Replace(arrC, UnaryPredicates.IsOdd, 1.0);
```