Custom Iterators

In this article, I'll discuss custom iterators, predicates, and generic methods. You can combine these three techniques to create small building blocks that plug together in a variety of ways to solve everyday development problems. Once you can recognize when to apply this pattern, you will be able to create a large number of reusable building blocks.

When we build software, we rarely work with one instance of a data type. Instead, most of our work centers on collections of data: **Lists**,**Arrays**,**Dictionaries**, or other collections. As a result, so much of our code appears in a series of loops:

```
foreach ( Thing a in someContainer)
   doWorkWith(a);
```

That's simple enough, and there's little point in trying to get more reuse from that simple construct. But most of our daily work isn't really that simple. Sometimes the iteration creates a new collection, or a new sequence. Sometimes the iteration should only affect some of the elements in a collection, not all them. Sometimes the iteration will remove some elements from the collection. With all the different variations, we're back to copying code and modifying it. That lowers our productivity. When you create custom iterators and predicates instead of copying and modifying loops, you decouple the common actions from the specific code. And, often, you will use the output from one iteration as the input to another. By using custom iterators to perform the iteration, you can save memory and increase performance as well.

Let's start with a simple sample, and modify that code to create a set of reusable custom iterators and predicates. Suppose you've been asked to print out a list of people you know from the New York City phone book. It is a contrived problem, but it exhibits the characteristics of many real-world problems. You need to process a large number of objects (the list of all entries in the New York City phonebook). You need to extract a subset of the properties of each object (the first and last names). Finally, you need to filter the input set to generate the output set (only the people you know). Your first attempt might look like this:

```
// First attempt:
public List<string> PeopleIKnowInNewYork()
{
   IEnumerable<PhoneBookEntrys> newYorkNumbers =
     PhoneBook.FindListFor("New York");
   List<string> peopleIKnow = new List<string>();
   foreach ( PhoneBookEntry ph in newYorkNumbers)
   {
      string name = string.Format("{0} {1}", ph.FirstName, ph.LastName);
     if ( RecognizePerson( name ) )
```

```
            peopleIKnow.Add(name);
    }
    return peopleIKnow;
}
```

This code does produce the proper output, but there's a lot to criticize. You've loaded every name in the New York phone book into one list. That's very wasteful. Chances are you know a very small percentage of people in New York. It is a big place, after all. However, by creating a local **List** of all the people in New York, you prevent the garbage collector from freeing any of those entries until the entire list is processed. At best, that's very wasteful. If you don't require a very large memory configuration, your application probably fails. Also, from a design standpoint, it falls short of expectations. Chances are that as this imaginary sample application grows, you will get other requests for similar, but not identical, features. For example, you may be asked to find everyone you called in the last six months, and print out the phone number you dialed. Right now, that produces another method that has almost the exact same contents, with one or two changes.

As a first step to creating more usable code, you can create a custom iterator that returns a sequence of names from a sequence of PhoneBookEntries. That would look like this:

```
IEnumerable<string> ConvertToNames(IEnumerable<PhoneBookEntry> list)
{
    foreach ( PhoneBookEntry entry in list)
        yield return string.Format("{0} {1}", entry.FirstName, entry.LastName);
}
```

This additional method changes your PeopleIKnow method to this:

```
// Second attempt:
public List<string> PeopleIKnowInNewYork()
{
    IEnumerable<PhoneBookEntry> newYorkNumbers =
        PhoneBook.FindListFor("New York");
    List<string> peopleIKnow = new List<string>();
    foreach ( string name in CovertToNames(newYorkNumbers))
    {
        if ( RecognizePerson( name ) )
            peopleIKnow.Add(name);
    }
    return peopleIKnow;
}
```

It's a little better. Now, anytime you get a new request that requires you to convert a set of phone entries to a set of strings, you've already got the method to do it. One more quick modification to the method saves you a lot of memory. If you examine the application, you'll almost certainly find that you never need the full list of names. You really only need to enumerate the list of names. So, you can try this modification and see if everything still compiles:

public IEnumerable<string> PeopleIKnowInNewYork()

That works, so you can change the**PeopleIKnowInNewYork** method to a custom enumerator method:

```
// Third attempt:
public IEnumerable<string> PeopleIKnowInNewYork()
{
  IEnumerable<PhoneBookEntry> newYorkNumbers =
    PhoneBook.FindListFor("New York");
  foreach ( string name in ConvertToNames(newYorkNumbers))
  {
    if ( RecognizePerson( name ) )
      yield return name;
  }
}
```

Let's stop a minute and consider what you've accomplished, and how you can use these techniques in other use cases and other applications. For this discussion, let's assume you've changed **PhoneBook.FindListFor()** to be an enumerator method as well.

You started with a pipeline that looked like this:

1. Create a list of every phone book entry in the New York phonebook
2. Examine every entry in that list
3. Create name from the phone book entry
4. If the name is recognized, add it to the output list

By changing methods to enumerators, you've created a pipeline that looks like this:

1. Read an entry from the phone book.
2. Create a name for that entry
3. If the name is recognized, return the name
4. repeat

That's a good start. This set of changes took away much of the memory pressure that this method placed on the system. There may only be one **PhoneBookEntry** and one string representation of a name in memory at one time. Any**PhoneBookEntry** objects already processed are eligible for garbage collection.

You can do better by introducing predicates, in the form of .NET delegates. The code you created is a little bit more reusable, but still suffers from being very specific to the operation at hand: finding recognized names from the New York Phone book. In order to make this code more reusable, you need to parameterize that specific portion of the algorithm. The problem, though, is that the specific portion of the algorithm is actually code. Luckily, the .NET Framework and C# have a way to introduce code as a parameter to a method: a delegate.

Start with your **ConvertToNames** method. With that as a base, you can build what we want: a method that transforms an input sequence into an output sequence of a different type. Here is the **ConvertToNames** method:

```
IEnumerable<string> ConvertToNames(IEnumerable<PhoneBookEntry> list)
{
   foreach ( PhoneBookEntry entry in list)
      yield return string.Format("{0} {1}", entry.FirstName, entry.LastName);
}
```

What you want is to pass the code 'string.Format(…)" as a parameter to the method. So you change the signature of**ConvertToNames** like this:

```
// A generic method to transform one sequence into another:
delegate Tout Action<Tin, Tout>(Tin element);
IEnumerable<Tout> Transform<Tin, Tout>(IEnumerable<Tin> list, Action<Tin, Tout> method)
{
   foreach( Tin entry in list)
      yield return method(entry);
}
```

This has some new syntax, but it's really fairly simple. The delegate definition defines the signature of any method that takes one input parameter and returns a single object of another type. Transform simple defines a pipeline that returns the output of the delegate for every object in the input sequence. It's the same thing as ConvertToName, but it can be used for any input type, any output type, and any algorithm that transforms one type into another.

You'd call that method like this:

```
// Fourth attempt:
public IEnumerable<string> PeopleIKnowInNewYork()
{
   IEnumerable<PhoneBookEntry> newYorkNumbers =
      PhoneBook.FindListFor("New York");
   foreach ( string name in Transform<PhoneEntry, string>(newYorkNumbers,
      delegate(PhoneEntry entry)
      {
         return string.Format("{0} {1}", entry.FirstName, entry.LastName);
      }))
   {
      if ( RecognizePerson( name ) )
         yield return name;
   }
}
```

There are a few new concepts here, so let's go over it in detail. This code is a little easier to understand if you change the structure just a bit. So here's the fifth version:

```
// Fifth attempt:
public IEnumerable<string> PeopleIKnowInNewYork()
{
   IEnumerable<PhoneBookEntry> newYorkNumbers =
      PhoneBook.FindListFor("New York");
   IEnumerable<string> names = Transform<PhoneEntry, string>(newYorkNumbers,
      delegate(PhoneEntry entry)
      {
         return string.Format("{0} {1}", entry.FirstName, entry.LastName);
      });
   foreach (string name in names)
   {
      if ( RecognizePerson( name ) )
         yield return name;
   }
}
```

The expression defining names defines a new enumeration over all the names harvested from the phone book. It defines the delegate method inline, making use of anonymous delegates. Note that I'm only creating a single line delegate here. I'd recommend against creating complicated methods as anonymous delegates. But, when you need to wrap an existing function call in a delegate, this syntax is much simpler for other developers to follow.

There's one last loop to convert to something more generic: The loop that checks for names you know. This one can be refactored into a well-known generic pattern:

```
delegate bool Predicate<T>(T inputValue);
IEnumerable<T> Filter<T>(IEnumerable<T> list, Predicate<T> condition)
{
   foreach (T item in list)
      if (condition(item))
         yield return item;
}
```

You should recognize this pattern by now. The Filter is just a pipeline that returns all members of a list that match the condition specified by the predicate. One last look at that PeopleIKnowInNewYork method shows how you can use it:

```
// Fifth attempt:
public IEnumerable<string> PeopleIKnowInNewYork()
{
   IEnumerable<PhoneBookEntry> newYorkNumbers =
      PhoneBook.FindListFor("New York");
   IEnumerable<string> names = Transform<PhoneEntry, string>(newYorkNumbers,
      delegate(PhoneEntry entry)
      {
         return string.Format("{0} {1}", entry.FirstName, entry.LastName);
```

```
    });
  return Filter(names, delegate(string name)
    { return RecognizePerson(name);} );
}
```

The final version is quite a bit different than what you started with, and some may argue that it's not that different. But it depends on how you apply it. The final version is structured around much more powerful building blocks. You built a generic method that filters a list of any type, as long as you supply the specific condition for a single item. You built a generic method that transforms a sequence of one type into a sequence of another type. You can leverage this same technique for other operations. This method samples an input sequence, returning every Nth item:

```
IEnumerable<T> NthItem<T>(IEnumerable<T> input, int sampleRate)
{
  int sample = 0;
  foreach (T aSample in input)
  {
    ++sample;
    if (sample % sampleRate == 0)
      yield return aSample;
  }
}
```

This method generates a sequence, based on some factory method you define:

```
delegate T Generator<T>();
IEnumerable<T> Generate<T>(int number, Generator<T> factory)
{
  for (int i = 0; i < number; i++)
    yield return factory();
}
```

And this method merges two sequences of the same type into a new type combining them:

```
delegate Tout MergeOne<Tin, Tout>(Tin a, Tin b);
IEnumerable<Tout> Merge<Tin, Tout>(IEnumerable<Tin> first,
  IEnumerable<Tin> second, MergeOne<Tin, Tout> factory)
{
  IEnumerator<Tin> firstIter = first.GetEnumerator();
  IEnumerator<Tin> secondIter = second.GetEnumerator();
  while (firstIter.MoveNext() && secondIter.MoveNext())
    yield return factory(firstIter.Current, secondIter.Current);
}
```

By separating the actions on a single object from the actions on a sequence of types, you created a set of more powerful building blocks to work with very large sets of sophisticated data types.

Summary

In this article, I showed you how custom iterators, delegates, and generic methods can be combined to create more powerful reusable building blocks for your applications. By applying these techniques yourself, you'll find many uses for these techniques, and you'll be more ready for C# 3.0, where many of these techniques enjoy more embedded language support.