Extracted from:

The ThoughtWorks Anthology 2

More Essays on Software Technology and Innovation

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

The ThoughtWorks® Anthology 2

More Essays on Software Technology and Innovation

Edited by Michael Swaine

The ThoughtWorks Anthology 2

More Essays on Software Technology and Innovation

Farooq Ali
Ola Bini
Brian Blignaut
James Bull
Neal Ford
Martin Fowler
Luca Grulla
Alistair Jones
Aman King
Patrick Kua
Marc McNeill
Julio Maia
Mark Needham
Sam Newman
Rebecca Parsons
Cosmin Stejerean

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

4.1 Collections

When it comes to understanding how a functional approach to problem solving can be used, one of the first things to consider is the way that we view collections.

The Transformational Mind-Set

The most interesting mental paradigm switch when learning how to program in a functional way is how you deal with collections.

With an imperative approach, you think about each item in the collection individually, and you typically use a `for each` loop when working with that collection.

If we take a functional approach to solving problems with collections, our approach becomes much more about viewing the collection as a whole—something that Patrick Logan refers to as a *transformational mind-set*.[2]

We look at the original collection that we have and then visualize how we want it to look once we've transformed it, before working out which functions we need to apply to the collection to get it into that state.

Original -> () -> () -> () -> Final

It closely resembles the pipes and filters architecture where the data moves through a pipe, and the filters are represented by the different functions that can be applied to that data.

Our approach to dealing with collections in this way is possible by using what Bill Six calls *functional collection patterns*.[3]

There are three main categories of operations on collections.

Map

The *map* pattern applies a function to each element in the collection and returns a new collection with the results of each function application (see Figure 1, *The Map Function*, on page 6). Therefore, if we want to get the first names of a group of people, we would write the following code:

```
var names = people.Select(person => person.FirstName)
```

rather than the following imperative equivalent:

2. http://www.markhneedham.com/blog/2010/01/20/functional-collectional-parameters-some-thoughts/#comment-30627
3. http://www.ugrad.cs.jhu.edu/~wsix/collections.pdf

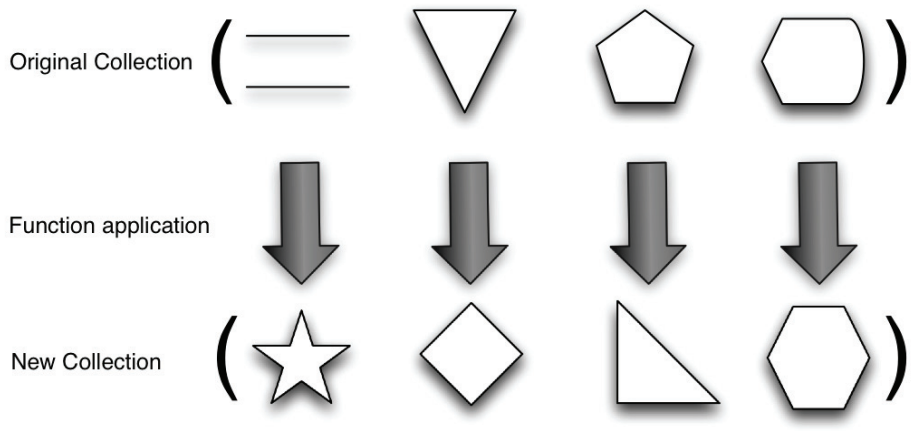
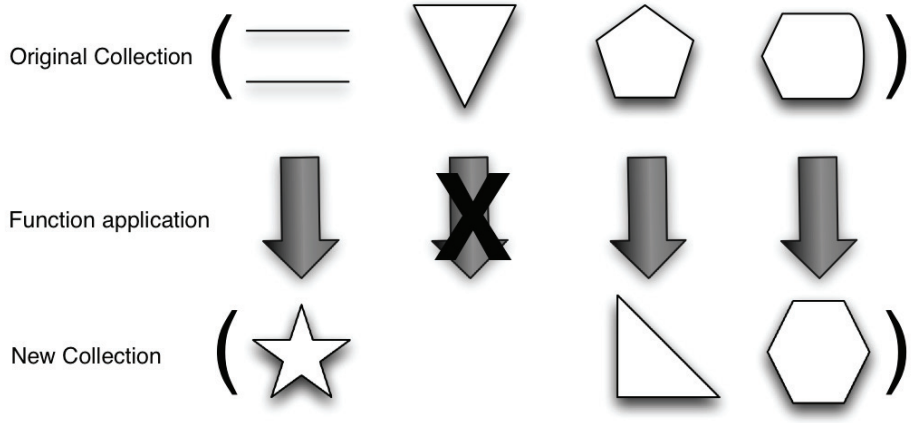


Figure 1—The Map Function

```
var names = new List<string>();
foreach(var person : people)
{
   names.Add(person.FirstName);
}
```

Filter

The *filter* pattern applies a predicate to each element in the collection and returns a new collection containing the elements that returned true for the predicate provided.



If we want to get only the people older than twenty-one years old, we would write the following code:

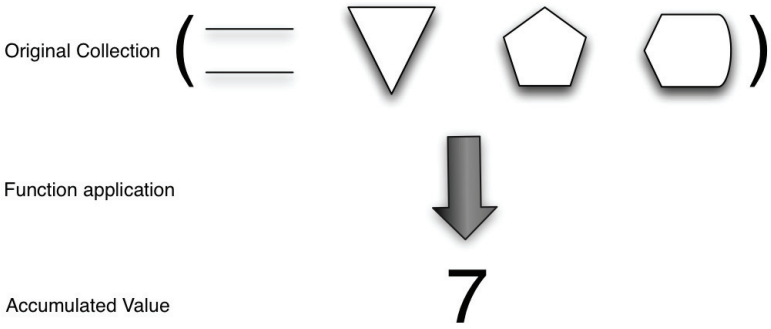
```
var peopleOlderThan21 = people.Where(person => person.Age > 21);
```

which is again simpler to read than the following imperative equivalent:

```
var peopleOlderThan21 = new List<Person>();
foreach(var person : people)
{
   if(person.Age > 21)
   {
      peopleOlderThan21.Add(person);
   }
}
```

Reduce

The *reduce* pattern converts a collection to a single value by combining each element in turn via a user-supplied function.



If we want to get the ages of a group of people, we would write the following code:

```
var sumOfAges = people.Aggregate(0, (sum, person) => sum + person.Age);
```

as compared to this:

```
var sumOfAges = 0
foreach(var person : people)
{
   sumOfAges += person.Age;
}
```

Embracing Collections

Once we get into the habit of applying functions to collections, we start to see more opportunities to use a collection where before we might have used a different approach.

Quite frequently I've noticed that we end up with code that more closely describes the problem we're trying to solve.

To take a simple example, if we wanted to get a person's full name, we might write the following code:

```
public class Person
{
    public string FullName()
    {
        return firstName + " " + middleName + " " + lastName;
    }

}
```

which works fine, but we could write it like this instead:

```
public class Person
{
    public string FullName()
    {
        return String.Join(" ", new[] { firstName, middleName, lastName });
    }
}
```

In this case, it doesn't make a lot of difference, and there's not that much repetition in the original version. However, as we end up doing the same operation to more and more values, it starts to make more sense to use a collection to solve the problem.

A fairly common problem I've come across is comparing two values against each other and then returning the smaller value. The typical way to do that would be as follows:

```
public class PriceCalculator
{
    public double GetLowestPrice(double originalPrice, double salePrice)
    {
        var discountedPrice = ApplyDiscountTo(originalPrice);
        return salePrice > discountedPrice ? discountedPrice : salePrice;
    }
}
```

But instead we could write it like this:

```
public class PriceCalculator
{
    public double GetLowestPrice(double originalPrice, double salePrice)
    {
        var discountedPrice = ApplyDiscountTo(originalPrice);
        return new [] { discountedPrice, salePrice }.Min();
    }
}
```

The second version is arguably easier to understand because the code reads as return the minimum of discountedPrice and salePrice, which perfectly describes what we want to do.

Don't Forget to Encapsulate

One unfortunate side effect of the introduction of the LINQ library and the consequent ease with which we can now work with collections is that we tend to end up with collections being passed around more than we would have previously.

While this isn't a problem in itself, what we've seen happen is that we'll get a lot of repetition of operations on these collections that could have been encapsulated behind a method on the object that the collection is defined on. The other problem with passing around collections is that we can do anything we want with that collection elsewhere in the code.

We worked on a project where it became increasingly difficult to understand how certain items had ended up in a collection because you could add and remove any items from the collection from multiple places in the code.

Most of the time, it's unlikely that the domain concept we're trying to model with a collection actually has all the operations available on the C# collection APIs. LINQ typically gets the blame when these problems occur, but it's more a case of it being used in the wrong place.

The following is a typical example of passing around a collection:

```
company.Employees.Select(employee => employee.Salary).Sum()
```

We could easily end up with the calculation of the employees' salaries being done in more than one place, and our problem would be increased if we added more logic into the calculation.

It's relatively easy to push this code onto the `Company` class.

```
public class Company
{
    private List<Employee> employees;
    public int TotalSalary
    {
        get
        {
            return employees.Select(employee => employee.Salary).Sum();
        }
    }
}
```

Sometimes it makes sense to go further than this and create a wrapper around a collection.

For example, we might end up with a Division that also needs to provide the TotalSalary of its employees.

```
public class Division
{
    private List<Employee> employees;
    public int TotalSalary
    {
        get
        {
            return employees.Select(employee => employee.Salary).Sum();
        }
    }
}
```

We can create an Employees class and push the logic onto that.

```
public class Employees
{
    private List<Employee> employees;
    public int TotalSalary
    {
        get
        {
            return employees.Select(employee => employee.Salary).Sum();
        }
    }
}
```

We've frequently seen a lot of resistance to the idea of creating classes like this, but if we start to have more logic on collections, then it can be quite a good move.

Lazy Evaluation

JavaScript sees extensive usage of its language. One problem that we can very easily run into when using iterators is evaluating the same bit of code multiple times.

For example, we might have the following code reading a list of names from a file:

```
public class FileReader
{
    public IEnumerable<string> ReadNamesFromFile(string fileName)
    {
        using(var fileStream = new FileStream(fileName, FileMode.Open))
```

```
        {
            using(var reader = new StreamReader(fileStream))
            {
                var nextLine = reader.ReadLine();
                while(nextLine != null)
                {
                    yield return nextLine;
                    nextLine = reader.ReadLine();
                }
            }
        }
    }
}
```

which is then referenced from our PersonRepository.

```
public class PersonRepository
{
    private FileReader fileReader;
    IEnumerable<Person> GetPeople()
    {
        return fileReader.ReadNamesFromFile("names.txt")
                         .Select(name => new Person(name));
    }
}
```

It's used elsewhere in our code like so:

```
var people = personRepository.GetPeople();
foreach(var person in people)
{
    Console.WriteLine(person.Name);
}

Console.WriteLine("Total number of people: " + person.Count());
```

The file actually ends up being read twice—once when printing out each of the names and then once when printing out the total number of people because the ReadNamesFromFile() method is lazy evaluated.

We can get around that by forcing eager evaluation.

```
public class PersonRepository
{
    private FileReader fileReader;
    IEnumerable<Person> GetPeople()
    {
        return fileReader.ReadNamesFromFile("names.txt")
                         .Select(name => new Person(name))
                         .ToList();
    }
}
```