C# toolkit™

Programming with the C# Toolkit

By Dong Nguyen

Abstract

With the evolution of the Internet to what it is today, companies have taken full advantage of the speed with which they can acquire the wealth of information that is available. Most likely, these companies employ software developers to come up with the means to process such huge quantities of information. The problem that developers often encounter when working with groups of information is that algorithm-specific logic can get littered throughout their code. To compound the problem, the enumerator provided by the .NET Framework is not sufficiently equipped for practical use. Recursion Software's C# Toolkit™ solves these problems by providing a set of containers, iterators, and algorithms to help developers cope with the increasing demands of an information-driven business.

Introduction to the Problem

Every developer usually encounters or works with groups of related data (which will be referred to as datasets from now on), and the size of those datasets can be very large. When it comes time to process that data, developers are left with only their wits as their only means to produce the logic and to apply the necessary functions/formulas step-by-step to acquire the desired results. Since the burden is placed on the developer, the code is often cluttered with custom algorithms throughout. In most occasions, the same or similar algorithm is used in several places. For example, the developer may write some code to count the number of items that are positive:

```
IEnumerator e1 = container.GetEnumerator( );
while ( e1.MoveNext( ) )
  if ( e1.Current > 0 )
    count++;
```

Elsewhere, a section of code counts the number of string objects:

```
IEnumerator e2 = container.GetEnumerator( );
while ( e2.MoveNext( ) )
  if ( e2.Current is typeof( String ) )
    count++;
```

Notice the amount of duplicated code in these examples. While the above code is trivial, combine this with more complex algorithms, and it becomes apparent that readability and maintenance become more difficult. Although this logic can be condensed into a method of the collection, this practice will only lead to a more bloated interface as more algorithms are added. A better way of dealing with this problem is to separate this responsibility to another class. In relation to the GRASP pattern (General Responsibility Assignment Software Pattern) conceived

by Craig Larman[1], this would fall into the Pure Fabrication section. The reason being is that even though the collection has all of the information (Expert Pattern), giving it too many and too varied responsibilities causes high coupling and low cohesion, whereas the opposite is preferred. The .NET Collections namespace was designed with these guidelines in mind. However, the .NET Framework, as feature-rich as it is, has no concept of or abstractions for commonly used algorithms by which developers can write cleaner, more easily maintainable code.

Along the same lines as working with datasets, the .NET Collections namespace supplies a class to traverse collections, called `IEnumerator`. This enumerator class has one property, `Current` and 2 methods, `MoveNext(-)` and `Reset(-)`. For situations where going across a collection to examine the elements is the only requirement, the standard enumerator class is sufficient. However for situations where modifying, place-holding, comparing enumerators, etc. is needed, the enumerator class just does not have the necessary functionality. These limited capabilities make it difficult for external algorithm classes to operate on the contents of the collections.

Solution

C# Toolkit solves these problems by providing a framework that promotes the separation of responsibility, the reuse of those facilities and sound programming practices, which enables developers to quickly build applications and easily maintain their code. With C# Toolkit, developers can immediately utilize and extend it to accommodate their needs. Like the code sample above, here is how the developer can make use of C# Toolkit algorithms:

```
count = Counting.CountIf( container, new PositiveNumber( ) );
```

and

```
count = Counting.CountIf( container, new Is( typeof( String ) ) );
```

By taking advantage of the C# Toolkit algorithms, there is less code, and the intent of its logic is clearer. There is no second-guessing when another developer comes along and tries to follow the code. The code thus produced is also more easily maintained. Let's say the developer brainstorms this complex algorithm and uses it in several places. If the developer finds out later that the algorithm needs tweaking, the change is only made in one place, contrary to the developer searching and changing each algorithm in many different locations.

In conjunction with the flexible framework, C# Toolkit provides 23 ready-to-use algorithms:

Applying

ForEach()	apply a function to every item in a range
Inject()	iteratively apply a binary function to every item in a range

Comparing

Equal()	check that two sequences match
LexicographicalCompare()	Lexicographically compare two sequences
Median()	return the median of three values
Mismatch()	search two sequences for a mismatched item

[1] Craig Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design,* Prentice Hall PTR, Upper Saddle River, NJ, 1998.

Copying

Copy()	copy a range of items to another area
CopyBackward()	copy a range of items backwards to another area

Counting

Accumulate()	sum the values in a range
AdjacentDifference()	calculate and sum the difference between adjacent values
Count()	count items in a range that match a value
CountIf()	count items in a range that satisfy a predicate

Filling

Fill()	set every item in a range to a particular value
FillN()	set n items to a particular value

Filtering

Reject()	return all values that do not satisfy a predicate
Select()	return all values that satisfy a predicate
Unique()	collapse all consecutive values in a sequence
UniqueCopy()	copy a sequence, collapsing consecutive values

Finding

AdjacentFind()	locate consecutive sequence in a range
Detect()	return first item that satisfies a predicate
Every()	return true if every item in a range satisfies a predicate
Find()	locate an item in a sequence
FindIf()	locate an item that satisfies a predicate in a range
Some()	return true if at least one item in a range satisfies a predicate

Heap

MakeHeap()	make a sequence into a heap
PopHeap()	pop the top value from a heap
PushHeap()	place the last element into a heap
SortHeap()	sort a heap

MinMax

MinElement()	return the minimum item within a range
MaxElement()	return the maximum item within a range

Permuting

NextPermutation()	change sequence to next Lexicographic permutation
PrevPermutation()	change sequence to previous Lexicographic permutation

Printing

Write()	prints a sequence to standard output
WriteLine()	prints a sequence to standard output followed by a newline
ToString()	returns a description of a sequence

Removing

Remove()	remove all matching items from a sequence
RemoveCopy()	copy a sequence, removing all matching items
RemoveCopyIf()	copy sequence, removing all that satisfy predicate
RemoveIf()	remove items that satisfy predicate from sequence

Replacing

Replace()	replace specified value in a sequence with another
ReplaceCopy()	copy sequence, replacing matching values
ReplaceCopyIf()	copy sequence, replacing values that satisfy predicate
ReplaceIf()	replace specified values that satisfy a predicate

Reversing

Reverse()	reverse the items in a sequence
ReverseCopy()	create a reversed copy of a sequence

Rotating

Rotate()	rotate a sequence by n positions
RotateCopy()	copy a sequence, rotating it by n positions

SetOperations

Includes()	search for one sequence in another sequence
SetDifference()	create set of elements in 1st sequence that are not in 2nd
SetIntersection()	create set of elements that are in both sequences
SetSymmerticDiffernce()	create set of elements that not in both sequences
SetUnion()	create set of elements that are in either sequence

Shuffling

RandomShuffle()	randomize sequence using random shuffles

Sorting

IterSort()	create iterators that will traverse a container in a sorted manner without reordering the container itself
Sort()	sort a sequence

Swapping

IterSwap()	swap the values indicated by two iterators
SwapRanges()	swap two ranges of items

Transforming

Collect()	return result of applying a function to every item in a range
Transform()	transform one sequence into another

Most of the algorithms can take a C# Toolkit function and/or predicate to slightly modify the algorithm's behavior. This yields numerous permutations on the already rich set of algorithms provided. The simplicity and modularity of the design of C# Toolkit makes it easy for developers to add their own algorithms, functions, and predicates.

C# Toolkit addresses the deficiencies of the standard enumerator by offering several iterator classes, which provide a richer set of semantics for container traversal and element manipulation without exposing the internal structure of the container class.

In addition, C# Toolkit extends the standard .NET Framework with a series of advanced containers, which implements the `IContainer` interface that exposes a richer set of semantics that the `ICollection` interface does.

Conclusion

In conclusion, the problems that developers will most likely encounter when working with collections are not trivial ones to solve; however, these problems can be overcome by making the most of the facilities provided by the C# Toolkit while, at the same time, help the developer write cleaner, clearer, and more maintainable code.

For additional technical information on Recursion Software products and programs or for information on how to order and evaluate Recursion Software technology, contact us today!

Recursion Software, Inc.
2591 North Dallas Parkway, Suite 200
Frisco, Texas 75034
1.800.727.8674 or 972.731.8800
www.recursionsw.com

RECURSION SOFTWARE, Inc.