Braces Considered Loopy

[From “The Communications of the ACM”, http://cacm.acm.org/magazines/2016/3/198861-acm-moral-imperatives-vs-lethal-autonomous-weapons/fulltext]

Braces Considered Loopy

The "naked braces" discussion, beginning with A. Frank Ackerman's letter to the editor "Ban 'Naked' Braces!" (Oct. 2015 [https://cacm.acm.org/magazines/2015/10/192375-ban-naked-braces/fulltext]), perhaps misses the forest for the trees, as a major reason for deeply nested expressions is the inability of most programming languages to handle arrays without looping. This shortcoming further compounds itself by contributing to the verbosity of the boilerplate required for such looping (and multi-conditional) constructs.

Jamie Hale's proposed solution in his letter to the editor "Hold the Braces and Simplify Your Code" (Jan. 2016)—including "... small and minimally nested blocks ..."—to the issue first raised by Ackerman pointed in a good direction but may remain lost in the forest of intrinsically scalar languages. Small blocks of code are good, but, in most languages, doing so merely results in a plethora of small blocks, pushing the complexity to a higher level without necessarily reducing it.

A more functional, array-based way of looking at problems can, however, reduce that apparent complexity by treating collections of objects en masse at a higher level.

Given most programmers' lack of familiarity with array-oriented programming, it is difficult for anyone, including me, to provide a widely comprehensible pseudocode example of what I mean by this, but consider the following attempt, based on the problem of invoking different code based on transaction size breakpoints (where "transactions" is a vector of transaction sizes)

```
Perform ( smallT(), largeT() )
BasedOn transactions > 1000
```

Ignoring the length discrepancy between the number of functions provided and the ostensible shape of the Boolean condition on which their selection is based, such a construct could easily be extended to additional breakpoints with something like this

```
Perform ( smallT(), largeT(), veryLargeT() )
BasedOn transactions > (1e3, 1e5)
```

For anyone interested in wrestling with a specific example of an array-based functional notation guiding my thoughts on this example, see http://code.jsoft-ware.com/wiki/Vocabulary/atdot.

Devon McCormick, New York, NY

An Example of Too-deeply-nested Code

[from http://www.codeproject.com/Articles/783225/An-alternative-introduction-to-Genetic-and-Evoluti]

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;

namespace Pfz.TravellingSalesman
{
      public static class RandomProvider
      {
            private static readonly Random _random = new Random();

            public static int GetRandomValue(int limit)
            {
                  return _random.Next(limit);
            }

            public static Location[] GetRandomDestinations(int count)
            {
                  if (count < 2)
                        throw new ArgumentOutOfRangeException("count");

                  Location[] result = new Location[count];
                  for(int i=0; i<count; i++)
                  {
…
```

[114 lines not shown]

```
                        using(var enumeratorIndex = toReplaceIndexes.GetEnumerator())
                        {
                              using(var enumeratorLocation =
availableLocations.GetEnumerator())
                              {
                                    while(true)
                                    {
                                          if (!enumeratorIndex.MoveNext())
                                          {

      Debug.Assert(!enumeratorLocation.MoveNext());
                                                break;
                                          }

                                          if (!enumeratorLocation.MoveNext())
                                                throw new
InvalidOperationException("Something wrong happened.");

                                          locations1[enumeratorIndex.Current] =
enumeratorLocation.Current;
                                    }
                              }
                        }
                  }
            }
      }
}
```