

# The Iterator Design Pattern

*brian d foy, comdog@panix.com*

## Abstract

The Iterator pattern separates the details of traversing a collection so that they can vary independently. Perl provides some of these parts already, although in some cases I need to provide my own implementations of them.

## 1 Introduction

Iterators as a design pattern are much more apparent as a pattern in languages that do not have aggregates as first class objects. Perl has lists, along with the variable types arrays and hashes, and it is trivial to go through any of these with built-in Perl structures like `foreach`, `each`, `map`, and so on. The file input operator, `<>`, iterates through the lines of the file. The `readdir()` and `glob()` functions iterate through filenames. In scalar context, the match operator with the global flag, `m//g`, iterates through matches.

In Perl I do not have to write classes to traverse an array, but I may need to write code to go through my own complex data structures because Perl's built-in control structures, like `foreach()`, `map()`, and `grep()`, do not define an interface that objects can use to control the iteration. They take a list, and I have to know what all of the elements of the list at the same time. A data structure might not even exist—the elements might be represented by an algorithm that determines the next element so that the object does not have to store all of the elements.

Iterators involve three parts: the *data*, the *iterator* that knows how to get the next item, and the *controller* which invokes the iterator. These may not always show up as distinct parts.

The iterator has to know at least two things: how to get the next element and when no more elements are available. Depending on the type of traversal, it may also need to know something about its state or the order it should follow. The controller uses this logic through some sort of interface, which may or may not be apparent, to interact with the data.

Since I separate each of these things when I use this pattern, I can easily change any one of them without affecting the others since they are *loosely coupled*. I reduce their dependence on other to give ourselves more flexibility and to improve the reusability of my code. For instance, I can change the iterator's traversal from depth-first to breadth-first, and the controller says the same since it uses the same interface. I can use the same controller for other iterators with the same interface, or the same iterator with other controllers, which gives me more choices if I decide I need to change something.

## 2 Types of iterators

Iterators come in two types: those where the something else—a distinct controller—controls the iteration, called *external iterators*, and those where the iterator controls itself, called *internal iterators*. Which one I

decide to use depends on what I need to do. As the implementor of an iterator I have to do about the same amount of work in either case, although other programmers benefit, or suffer, from which one I decide to use.

## 2.1 Internal iterators

With internal iterators, I tell a method or function to perform some operation on each element of a collection because I already know that I will have to visit most or all of the elements, and as long as I do that I do not care how it happens. Internal iterators often combine the controller and iterator aspects into a single thing which simplifies the life of the programmer who uses the iterator in his code.

The `map()` and `grep()` built-in functions use internal iterators. They go through all of the elements in the data independent of our control because they combine the iterator and controller. I give these functions a bit of code that they apply to each of the elements in a list. The `map()` function returns a list of values based on the original list, and the `grep()` function returns a list of values from the original list that satisfied a condition. The controller and iterator parts are part of the perl core. I do not have to tell these functions how to get the next element in the data, when they have gone through all of the elements, or that they should move on to the next element.

---

Code Listing 25: Perl's built-in `map` and `grep`

---

```
1 my @squares = map { $_ * $_ } 0 .. 10;  
2  
3 my @odds = grep { $_ % 2 } 0 .. 100;
```

---

When I use the `File::Find` module, I let its `find()` function iterate through the directories on its own, and it applies the callback function to each file as it finds it.

---

Code Listing 26: `File::Find` internal iterator

---

```
1 #!/usr/bin/perl  
2 use File::Find ();  
3  
4 File::Find::find( { wanted => \&wanted }, '.' );  
5  
6 sub wanted  
7     {  
8     print "$File::Find::name\n" if /\.*\.\tex\z/s;  
9     }  
}
```

---

`File::Find` knows how to get to the next element because it keeps track of its place in the filesystem and makes repeated calls for directory contents. When it asks for more files and the filesystem tells it that no more exist, `File::Find` knows it is done. As the programmer, I do not know any of this though. Once I start the `find()` function, it moves from element to element on its own.

Modules which implement composite data structures can define methods to iterate over all of their elements to perform an operation, such as a callback function or a Visitor object. The `Netscape::Bookmarks` module represents the information in a Netscape (and now Mozilla, too) bookmarks file, usually stored in HTML on a local computer, in a data structure so I can do interesting things with the data such as spell-checking,

re-arranging, importing new links, converting formats, or link checking. On the insides it is a collection of different objects from the Netscape::Bookmarks classes Category, Link, Separator, and Alias.

The Netscape::Bookmarks::Category module provides a recurse() function that applies a call\_back routine to every element in the current category and below, as well as an introduce() routine that passes its elements one-by-one to a Visitor object. The module handles the details of the iteration and the controller for us. Once I start the iterator it goes to completion on its own. Code listing 27 shows how little work the application programmer needs to do to traverse the entire Netscape::Bookmarks structure—indeed, that is the point. Not only does the programmer need to write only a couple lines of code, but if I change the mechanics or the implementation, the programmer does not have to change his code.

---

Code Listing 27: Netscape::Bookmarks internal iterators

```
1 use Netscape::Bookmarks;
2
3 my $bookmarks = Netscape::Bookmarks->new( $file );
4
5 $bookmarks->recurse( \&call_back );
6
7 $bookmarks->introduce( $visitor );
```

---

The recurse() and introduce() methods define a traversal order, and that order is not important to me as long as I get a chance to process each element.

Classes define internal iterators to handle tasks that to operate on every element of the collection in a uniform matter. The iterator treats all of the elements the same way and when the iterator is done with one element it automatically moves on to the next one. I create methods like recurse() when I do not want to do a lot of work at the application level. My scripts which use the module obey the interface, and if I find a better way to do it, the applications do not need to change<sup>1</sup>.

## 2.2 External iterators

Most programmers use external iterators all the time without even knowing it. When I provide the controller and decide when to move on to the next element, I use an external iterator. Since the list is a fundamental Perl concept, Perl naturally has a lot of features to work with lists, and a lot of external iterator idioms.

To go through the elements of a list, I can use the foreach() control structure as an external iterator. I control the iterator because I have the ability to skip items (with next), stop the iteration (with last), or reprocess the current element (with redo). The foreach() controller does not move onto the next element until I let it, which I might do implicitly by not telling it to do something else.

---

Code Listing 28: External iterations with foreach()

```
1 foreach my $url ( @urls )
2   {
3     next if $link->scheme ne 'http';
4     redo if $link->domain eq 'www.perl.org';
5     last if $link->query =~ m/foo/;
6   }
```

---

<sup>1</sup>You may see a bit of the Facade design pattern here.

Some collections do not exist as lists and so I must explicitly access one element at a time. The `while()` control structure controls the iteration by repeatedly fetching the next element to process. In this case, I have to know how to get the next element. The DBI interface returns rows from a record set one at a time through the `fetchrow_array()` method. The DBD driver knows how to fetch the next element, and `fetchrow_array()` returns false when I have fetched all of the records, signalling the end of the iteration. I can decide to stop the iteration at any point, even if I have not fetched all of the records.

---

Code Listing 29: The DBI external iterator

---

```
1 use DBI;
2
3 my $dbh = DBI->new(...);
4 my $sth = $dbh->prepare(...);
5
6 while( my @row = $sth->fetchrow_array ) {
7     ... }
```

---

## 2.3 One controller, multiple iterators

Since I control external iterators, I can use more than one iterator at the same time. If I want to compare two files, for instance, I can use the line input operator on two different filehandles at the same time. Each time I go through the while loop I get the next item from each of the iterators. In code listing 30 the while controller uses two iterators.

---

Code Listing 30: One controller, multiple iterators

---

```
1 while( my ( $old, $new ) = (scalar <OLD>, scalar <NEW>) ) {
2     ... }
```

---

## 2.4 One iterator, multiple controllers

I do not have to use the same control for all parts of the iteration. In code listing 31 I read the first line of standard input using the line input operator (the iterator) with an assignment (the controller, if you will) to a scalar variable. Perhaps this line represents the column headings in a flat-file database. After that, I read in the next ten lines with a different controller, the `while()` loop, after which I go through the remaining lines with `grep()`. I can only do this because I can decide when to move on to the next element.

---

Code Listing 31: Multiple controllers

---

```
1 my $titles = <STDIN>;
2
3 my $count = 0;
4 while( <STDIN> ) {
5     last if $count++ >= 10;
6     ... }
7
8 my @lines = grep { /Perl/ } <STDIN>;
```

---

## 2.5 Working with data not in memory

I also use external iterators when all of the of the data cannot or should not be in memory at the same time. If I work with a tied DBM hash, my hash represents possibly large numbers of keys and values stored on disk. Since the elements of the hash are not in memory, I save space. If I use the keys() or values() functions those potentially large numbers of keys or values are now stored in memory, negating my savings. The each() iterator fetches one key-value pair at a time.

---

Code Listing 32: Iterating through a DBM file

---

```
1 while( my($key, $value) = each %DBM )
2     {
3     $sum += $value;
4     }
```

---

This is the same idiom as reading a file line-by-line rather than all at once. Since the filehandle potentially delivers more memory than our program can handle or more RAM than our hardware has, most people recommend you read the file one line at a time, like I do in code listing 33. The earlier DBI example in code listing 29 does the same thing.

---

Code Listing 33: Iterating through a DBM file

---

```
1 while( <FILE> )
2     {
3     ...
4     }
```

---

## 3 Iterator interfaces

Once I decide that I need to create my own iterator, I have to design an interface for it. The design pattern only shows me the general solution, so I have to look at the specific problem to see how I can apply the general pattern.

### 3.1 Object methods

Some modules save memory by computing elements only when needed, or fetch data on request from remote sources, like my earlier DBI example in code listings 29 or 32. In these cases the object has a method to return the next element.

The Set::CrossProduct module lets me deal with all of the combinations of elements from two or more sets. For instance, for the two sets (a,b) and (1,2) I get the combinations (a, 1), (a, 2), (b, 1), and (b, 2). The number of combinations is, at most, the product of the number of elements in each set, which means that the number of elements can be very large for even a small number of moderately sized set. Five sets of five items has over 3000 combinations. In code listing 34 I get back all of the combinations at once and store them in @combinations, meaning that I potentially use up a lot of memory even though I later go through the combinations sequentially in the foreach() loop. This has the same problems as reading an entire file into an array.

Code Listing 34: All combinations at once

```

1 use Set::CrossProduct;
2
3 my $cross_product = Set::CrossProduct->new( [ [qw(a b)], [ 1, 2 ] ] );
4
5 # get all combinations at once
6 my @combinations = $cross_product->combinations;
7
8 foreach my $item ( @combinations )
9     {
10        print "The combination is @$item\n";
11    }

```

Set::CrossProduct does not store the combinations in memory though. It simply stores the sets and keeps track of which combination it needs to make next. The combinations() method in code listing 35 has to create the list for me. Set::CrossProduct provides a next() method, the iterator, which lets a controller fetch the next value. It is an external iterator, so I need to provide the controller to make the iterator move from one element to the next so I only have to store one combination at a time. In code listing 35 I use a while() loop to repeatedly fetch the next combination—each time testing the return value of the next() method to see if it returned a combination.

Code Listing 35: Iterate through successive combinations

```

1 use Set::CrossProduct;
2
3 my $cross_product = Set::CrossProduct->new( [ [qw(a b)], [ 1, 2 ] ] );
4
5 while( my $item = $cross_product->next )
6     {
7        print "The combination is @$item\n";
8    }

```

### 3.1.1 When has the iterator finished?

How do I know when no more elements are available? The interface has to signal to the controller that the iterator has gone through all of the elements and that the controller should not ask for any more.

I can return a false value but does not always work. The line input operator, for instance, uses undef to signal the end of input. That means it does not use all the false values for this signal since 0 and the empty string are not undefined. Any value besides undef comes from the data source and is an element of the iteration. In code listing 36 I test specifically for the undef value to see if the input is finished.

Code Listing 36: Line input operator in while

```

1 while( defined( $line = <STDIN> ) ) {
2     ... }

```

Perl has a special idiom for this if I use the default variable \$\_. I could write it out to look the same as code listing 36 but with \$\_, or I can write in much more simply as in code listing 37 which does the same thing.

The `while()` condition tests to see if the item is defined, not that it is true. This is a special case only for when I use the line input operator with `$_` in the `while()` condition.

---

Code Listing 37: Test for a defined value, not a false one

```

1 while( <STDIN> ) { # really while( defined( $_ = <STDIN> ) )
2     ... }

```

---

What if `undef` value is a valid value? I cannot use it to signal the end of the iteration. I might be able to use another value that does not cannot appear in the data, but if any value is valid, I cannot use inspection to decide what to do<sup>2</sup>.

I can design an iterator which has another method which tells me the state of the iteration. I check this method before I attempt to fetch the next element to see if any more elements are available, and if none are, the controller knows to stop. In code listing 38, if the `has_more_elements()` method returns false I stop the iterator.

---

Code Listing 38: Check if more items are available

```

1 while( $iterator->has_more_elements )
2     {
3     $item = $iterator->next;
4     ...;
5     }

```

---

More Perl-like methods work too. I can always return a reference to the data instead of the data itself. Even a reference to a false value is true since I test to determine if the variable is a reference instead of checking its value. The iterator returns a non-reference to signal that no more elements are available. The interface is the almost the same as code listing 38 since a reference is always true, even if the data it points to would evaluate to false.

If the `next()` method always returns a reference, perhaps an array reference, I can tell the difference between `undef` and the anonymous array of one element that contains the `undef` value. The values `[ undef ]`, a reference, and `undef`, are different. Code listing 39 loops until the `next()` method returns any false value since references are always true.

---

Code Listing 39: Return a reference

```

1 while( my $ref = $iterator->next ) # [ undef ] works, undef doesn't
2     {
3     my @items = @$ref;
4     ...
5     }

```

---

## 3.2 Custom controllers

The `Object::Iterate` module defines some controllers for these sorts of interfaces so I can interact with the object just like I do with lists for `foreach()`, `map()`, and `grep()`. It defines `iterate`, `igrep`, and `imap` which

<sup>2</sup>Mark Jason Dominus calls this the Semi-predicate Problem

look almost just like the perl built-in functions, but takes an object that can respond to a couple of special method names. Code listing 40 shows the `iterate()`, `imap()`<sup>3</sup>, and `igrep()` functions.

---

Code Listing 40: Object::Iterate controllers

---

```

1 use Object::Iterate qw(iterate igrep imap);
2
3 iterate { print "$_\n" } $some_object;
4
5 my @output = imap { ... } $some_object;
6
7 my @filtered = igrep { ... } $some_object;

```

---

Each of these functions goes through all of the elements of the object through the object's interface. Without hints from the object, the `Object::Iterate` module uses the special object methods `__next__` and `__more__` to get the next element and determine if more elements exist. The object's class has to implement these methods itself, and the three controllers work with any object that follows the interface. The functions act as internal iterators just like the example in Section 2.1. Once I start them they go through the entire structure without further control from me.

Code listing 41 shows the implementation for the `iterate()` function. It takes an anonymous subroutine as its first argument which lets it mimic the syntax for `map {}`<sup>4</sup>. The object over which it will iterate is the second argument. In lines 5-8, `iterate()` ensures that the object has the right special methods. In the `while()` loop, `iterate()` does the same thing as in code listing 38.

---

Code Listing 41: Object::Iterate::iterate

---

```

1 sub iterate (&$)
2     {
3     my( $sub, $object ) = @_;
4
5     croak( "iterate object has no $Next() method" )
6         unless UNIVERSAL::can( $_[0], '__next__' );
7     croak( "iterate object has no $More() method" )
8         unless UNIVERSAL::can( $_[0], '__more__' );
9
10    while( $object->__more__ ) {
11        local $_;
12
13        $_ = $object->__next__;
14
15        $sub->();
16    }
17 }

```

---

<sup>3</sup>Mark Jason Dominus came up with similar methods for his "Iterators and Generators" talk, but implemented them differently. For awhile I felt bad about choosing "imap" for a name since its also a well-known protocol, but after I saw that he choose the same name, I did not feel so bad.

<sup>4</sup>See the Prototypes section in `perlsub`

### 3.3 Closures

I do not necessarily need modules and classes to create iterators either. I can use a closure to hold all of the information. If I want to iterate over odd numbers, an infinite series which I cannot ever completely store in memory, I can create a closure that returns the next odd number each time I call it. It maintains its own state and I avoid all of the overhead of method lookups.

---

Code Listing 42: Closures as iterators

---

```

1 my $odds = do { my $next = -1; sub { $next += 2; return $next } };
2
3 while( my $number = $odds->() )
4     {
5         print "The next number is $number\n";
6     }

```

---

Some people call closures “inside-out object”. Objects are data with behavior while closures are behavior with data, so they make handy iterators. I combine the data and iterator portion to create the closure. Each time I dereference the closure, I get back the next value. The closure comprises the iterator and the data portion, while I supply the controller.

### 3.4 Tied scalars

Tied scalars must have a FETCH method, but nothing specifies what I have to do or which data I have to return with that method. The Tie::Cycle module ties an anonymous array to a scalar so that each time I access the scalar’s value, I get the next item from the array, and when I get to the end of the array it goes back to the beginning. The controller is the use of the scalar on the right-hand side of an expression, the FETCH method defines the iterator, and the anonymous array stores the data. In this case, the tied scalar combines the iterator and the data, although I still provide the controller because I use program logic to decide when to access \$colors even though I do not use an explicit controller.

I initially created Tie::Cycle to handle alternating colors in rows of HTML tables. I grew weary of creating bugs when I changed the colors or their number, and the amount of distracting code that had to go into calculating an index that stayed within the bounds of the defined elements of an array. All I wanted was the next color, and I wanted that to be simple. Tie::Cycle handles the annoying parts for me, and I can reuse it wherever I need it. Code listing 43 shows a typical use to shade rows of HTML tables with varying levels of gray. Each time I access the tied variable \$colors, on line 7, the Tie::Cycle module advances along the anonymous array I gave it as an argument on line 3.

---

Code Listing 43: Tie::Cycle

---

```

1 use Tie::Cycle;
2
3 tie my $colors, 'Tie::Cycle', [qw(aaaaaa cccccc ffffff)];
4
5 foreach my $row ( @rows )
6     {
7         my $row_color = $colors;
8
9         print <<"HTML";

```

```

10 <tr>
11     <td bgcolor="$row_color">
12     ...
13     <td>
14 </tr>
15 HTML
16     }

```

---

### 3.5 Tied filehandles

I can attach almost any data to a filehandle with a tie. I have to implement some of the functionality myself, such as determining what the next “line” is, but once I have done that little bit of work I can use all of Perl’s filehandle iteration framework. This can be especially beneficial to new programmers who can work with filehandles but have yet not used Perl’s more advanced features.

In code listing 44 I tie a normal scalar to an input filehandle so I can read the scalar line-by-line or one character at a time just as if I were reading from a real file. Code listing 45 shows its use in a program. The READLINE function defines how to read a line (or several lines in list context), and the GETC function defines how to read one character. Perl uses READLINE when I use the file input operator <> and GETC when I use the `getc()` function. In each case, the module removes the piece that I read, so our scalar gets shorter and shorter (unless I add to it by some other means, which I might want to do if I create an in-memory buffer).

I can decide how to read my lines. In this case, I avoid an annoying chomp by not returning the current value of the input record separator, `$/` (a newline by default, but maybe something different) which in code listing 44 I assign to `$EOL` in line 14 and use in the regular expression, although outside the memory parentheses, in line 21.

Code Listing 44: Treat a scalar as a file

```

1 package Scalar::Iterator;
2
3 sub TIEHANDLE
4     {
5     my( $class, $text ) = @_;
6
7     bless \$text, $class;
8     }
9
10 sub READLINE
11     {
12     my $self = shift;
13
14     my $EOL = $/;
15     if( length $$self > 0 )
16     {
17     my @lines = ();
18
19     while( length $$self > 0 )
20     {
21     $$self =~ s/(.*?)$EOL||s;
22     print "Matched $1\n";

```

```

23         push @lines, $1;
24         last unless wantarray;
25     }
26
27     return wantarray ? @lines : $lines[0];
28 }
29 else
30 {
31     return; # undef signals the end
32 }
33 }
34
35 sub GETC
36 {
37     my $self = shift;
38
39     return $1 if $$self =~ s|(.)||s;
40
41     return; # undef signals the end
42 }

```

Code Listing 45: Using a scalar as a file

```

1 use Scalar::Iterator;
2
3 my $data = ...;
4
5 tie *MY_DATA, 'Scalar::Iterator', $data;
6
7 my $line = <MY_DATA>;
8 print "Got one line [$line]\n";
9
10 my $char = getc( MY_DATA );
11 print "Got next character [$char]\n";
12
13 print "Got rest of lines:\n", <MY_DATA>;
14

```

I can change the way that I go through the scalar. I can change what I mean by “line” and “char” to be something else. After all, the computer does not really know what these things are. In code listing 46, I change `READLINE` to read the next sentence, and `GETC` to read the next word. This is more complicated than it sounds if I wanted to do this to arbitrary text, so I have to do more work than I do for the general case.

Ever wanted to put a regular expression into `$/`? Well, now I can. In code listing 46 I conveniently used `$EOL` as the end-of-line marker in my example, and I put it at the end of my regular expression in `READLINE`. If I put regular expression special characters in there, the `s///` operator will interpret them as regular expression sorts of things. In this case I think I have reached the end of the sentence when I run into the next punctuation character in the class `[.!?]`. This time I include the end-of-record marker, the sentence ending punctuation, with the sentence. At the same time I collapse consecutive whitespace to a single linear space.

I have to make a minor change to make GETC read words. For this example, I pretend that words only have alphabetic characters and ignore special cases like contractions, abbreviations, and hyphenated words. In that case, GETC only has to return the next sequence of letters while skipping over non-alphabetic characters it finds first.

My data has not changed. It still can be anything that I like, but I can easily change how I go through it, since all the bits of the iterator stay separate from the object (not really an instance, in this case). If I decide to change how I go through the object, the iterator is the only thing that changes. I can even define several different iterators and use them at the same time if I like. I do not have to do much more work to turn our sentence reader into a paragraph reader, and so on. Tied filehandles have infinite uses as iterators, but beware—tied filehandles can be slow. From ease-of-use, flexibility, and speed, I only get to choose two.

Code Listing 46: Sentences and words

```

1 sub READLINE
2   {
3     my $self = shift;
4
5     my $EOL = '[.!?]';
6     if( length $$self > 0 )
7       {
8         my @lines = ();
9
10        while( length $$self > 0 )
11          {
12            $$self =~ s|(.?*$EOL)||s;
13            my $sentence = $1;
14            $sentence =~ s/\s+/ /g;
15            push @lines, $sentence;
16            last unless wantarray;
17          }
18
19        return wantarray ? @lines : $lines[0];
20      }
21    else
22      {
23        return; # undef signals the end
24      }
25  }
26
27 sub GETC
28   {
29     my $self = shift;
30
31     return $1 if $$self =~ s|^[a-z]([a-z])||i;
32
33     return; # undef signals the end
34   }

```

## 4 Perl Modules which represent iterators

Besides the modules I used as examples, several other modules express the Iterator design pattern.

### 4.1 XML::\*, HTML::\*

Some of the XML and HTML modules represent stream-based parsers. I give them a big chunk of text, and the module breaks it into pieces and gives me the chance to interact with the pieces. The parsers act as the iterators and the controllers, while I supply behavior for the items they encounter. The parsers know how to get the next item.

### 4.2 File::ReadBackwards

The File::ReadBackwards is a simple iterator that gives me the lines from a file one at a time, only starting at the end and working its way to the beginning. I can use its object interface or its tied filehandle interface. I supply the external iterator, and the module knows how to get the next item.

### 4.3 Tie::DirHandle

The Perl built-in function `readdir()` is an iterator. In scalar context it gives me the next filename from the directory handle, and in list context give me back all of the filenames. It defines the logic of the traversal and I supply the controller. This modules hides the `readdir()` so I can use the filehandle iterators to interact with the directory handle. I get the next filename with the line input operator instead of `readdir()`.

### 4.4 Tie::IxHash

Normally, hashes do not preserve the order of the elements I add to them. Perl stores the key-value pairs in a, well, hash tree, for easy lookup. Several modules, including Tie::IxHash, remember the order of hash operations, including addition of keys, so I can get the keys back in the order that I added them. It uses a tied hash to define the logic of traversal, and I can then use the standard Perl external iterators idioms to traverse the hash.

## 5 Conclusion

The Iterator design pattern has three parts: the data, the iterator, and the controller. In most cases Perl supplies the iterator and controller with its built-in functionality. In some cases, I have to write my own iterator to decide how to get the next item in the data and to decide when the iteration is complete. As with other design patterns, the implementation is just an expression of the pattern, and there is more than one way to do it. Which way I choose depends on my particular problem.

## 6 References

*Design Patterns*, Erich Gamma, Richard Helm, Ralph Johnson, Jon Vlissides, Addison Wesley, 1995.

## 7 About the author

brian d foy is the publisher of *The Perl Review* and a Perl trainer for Stonehenge Consulting Services, <http://perltraining.stonehenge.com>.