Loop Patterns

Authors

```
        Owen Astrachan                      Eugene Wallingford
  Department of Computer Science      Department of Computer Science
    Duke University                    University of Northern Iowa
       Durham, NC 27708                   Cedar Falls, IA 50614
        ola@cs.duke.edu                    wallingf@cs.uni.edu
```

Background

There are many ways to look at patterns. An especially useful way to think of patterns is as a tool for teaching. We don't use patterns blindly; we learn them. Patterns are all about learning successful techniques, understanding when and how to use them. At ChiliPLoP'98, a small group of computer science educators gathered to think about and write patterns appropriate for novices learning to program in the first two years of undergraduate instruction. (For more information or to become involved in this ongoing project, visit the elementary patterns web page.)

The patterns contained here were first discussed and written as a part of the ChiliPLoP workshop. They are, we hope, the beginnings of what will be a pattern language for helping novice programmers construct loops. Writing loops is the basis for many of the problems students attempt to solve when writing programs in Algol-like languages. (In functional languages such as Scheme, recursion is typically used for repetition instead of loops. See, for example, Roundabout, a pattern language for recursive programming that was workshopped at PLoP'97.)

We focused our initial efforts on identifying specific problems that students encounter when learning to write loops. This lead us to several specific patterns. At this point, they are only rather loosely related, but we hope that they serve as a useful starting point for a more extensive documenting effort.

Linear Search

You are working with a collection or stream of objects.

How do you find an object that meets a specific condition?

Suppose that you have a set of students, and you would like to find the first student with an "A" average. In the worst case, you will look at the whole collection before finding your target. But, if you find a match sooner, you would like to terminate the search and work with the object that you found.

Therefore, construct a Process All Items loop, but provide a way to exit the loop early should you find your target sooner.

Express your student search as:

```
for (i = 0; i < students.size(); i++)
    if (student[i].grade().isAnA())
       break;

// process student[i], who has an A
```

If it is possible that you will not find your target, be sure to do a Guarded Linear Search.

Guarded Linear Search

You are doing a Linear Search.

How do you handle the possibility that no target may be found?

What happens if it is possible that no student has an "A" average? Your loop will look at the entire collection and still not find a target. But the code that follows the loop assumes that student[i] is the target.

Therefore, follow the loop with an Alternative Action that treats the "not found" situation as a special case.

Express your student search as:

```
for (i = 0; i < students.size(); i++)
    if ( student[i].grade().isAnA() )
       break;

if (i < students.size())              // found it!
   // process student[i], who has an A
else
   // handle the exception
```

[Resulting context...]

Process All Items

You are writing code that manipulates a collection.

How do you process all of the items in the collection?

The collection might be an array, a bag, a hashtable, a stream, or a file--in general, items stored in some structure and made accessible to the programmer through some collection/iterator interface. You want to process all of the items in an identical manner.

You may need to process all of the items because in the worst case all items must be processed (Linear Search), or because all items must be processed even in the best case, in order to ensure correctness (Extreme Values).

There are two kinds of process-all-items loops: a definite loop that uses indexing when the number of indexable items in a collection is known in advance, and an iterating loop when an iterating interface is used to access the items in a collection sequentially.

Therefore choose the appropriate loop and loop pattern for processing all the items in the collection once.

Definite Process All Items

You are writing a Process All Items loop for an indexable collection, the number of items in the collection can be determined simply (in constant time) and indexing is constant time.

How do you access every indexable item?

The collection might be a vector or a string where the number of elements in the collection can be determined by a function call, or the number of items might be another variable associated with the collection, e.g., as a parameter to a function or a field in a class.

Therefore use a Definite Process All Items for loop to touch evey element in the collection.

Example

The items are stored in a vector. Use a definite loop to process all the items, even for algorithms like Linear Search. In C++:

```
for(int k=0; k < v.size(); k++)
{
     process v[k]
}
```

Iterator Process All Items

You are writing a Process All Items loop for a collection that uses an iterator interface or when constant time indexing of the collection is not possible.

How do you access every item in the collection?

The items may be stored in a collection with a standard iterator interface, e.g., a Java enumeration.

The items may be stored in a simple linked list where there is no explicit iterator interface, but (iterating) links must be followed.

Therefore use an iterator process all items while loop that tests the iterator to see if it is finished.

Examples

In Java the items may be accessible via the standard Java 1.1 Enumeration interface.

```
    Hashtable table = new Hashtable();
    // code to put values in table

    Enumeration e = table.keys();
    while (e.hasMoreElements())
    {
        process(table.get(e.nextElement()));
    }
```

In C++ processing a linked list requires an iterating pointer variable:

```
   Node * ptr = first;
   while (ptr != NULL)
   {
       process(ptr->info);
       ptr = ptr->next;
   }
```

The Iterator pattern is widely known from the GOF patterns book, but is mentioned as an idiom in Coplien's Advanced C++

Loop and a Half

You are writing a Process All Items loop.

How do you structure the loop when the number of items in the collection isn't known in advance and the items are provided one-at-a-time?

For example, you might be reading values from the keyboard or from a file. In general, you are processing all items until a sentinel is reached. The sentinel may be end-of-file, a valid value for the type of data being read, or an exception.

The loop requires priming to read/get the first value, a loop guard to test for the sentinel, and another read/get inside the loop. This leads to duplicate code for the read/get and makes the loop body harder to understand since it turns a read-and-process loop into a process-and-read loop.

Therefore, use a while-true loop with a break in the loop body to avoid duplicate code and to match your intuition that the loop is a read-and-process loop.

For example, consider pseudocode for a sentinel loop:

```
    read value
    while (value != sentinel)
        process value
        read value
```

This can be rewritten as

```
while (true)
    read value
    if (value == sentinel) break;
    process value
```

Although the break is essentially a goto and thus violates a canon of structured programming, it results in code that it easier to develop Roberts. Because there is no code duplication the code should be easier to maintain and easier to understand --- see Assign Variables Once and Local Variables Reassigned Above Their Uses.

[Resulting Context: Patterns for writing (compound) Boolean expressions.]

Polling Loop

You are writing a program that asks the user to enter a data value.

How do you poll until the user enters a valid data item?

For example, suppose that we want the user to enter a legal grade, between 0 and 100 inclusive.

Many languages provide a `do...while...` construct that allows direct testing of a value after entry. For example:

```
do
{
   cout << "Enter a grade between 0 and 100, inclusive: ";
   cin  >> grade;
}
while (grade < 0 || grade > 100)
```

This solution gives the same prompt on all requests, so the user may be confused by the repetition. The test is negative, which requires students to use DeMorgan's laws to write the condition.

Instead, you could follow the data entry with an explicit validation loop:

```
cout << "Enter a grade between 0 and 100, inclusive: ";
cin  >> grade;

while (grade < 0 || grade > 100)
{
   cout << "Sorry!  That is an illegal value." << endl;
   cout << "Enter a grade between 0 and 100, inclusive: ";
   cin  >> grade;
}
```

This solution replicates the prompt code. We can lessen the negative effect by making the prompt a procedure call. But, the test is still written in the negative, which is difficult to do--and undo, for the reader who wants to know what the legal values are.

Both of these solutions stop processing when the negative condition fails, and the prompting sequence is difficult.

Therefore, use a loop and a half. Write a positive condition that stops the repetition when the user enters a legal value.

So, you might solve your grade-entering as:

```
while (1)
{
   cout << "Enter a grade between 0 and 100, inclusive: ";
   cin  >> grade;

   if (grade >= 0 && grade <= 100)
      break;

   cout << "Sorry!  That is an illegal value." << endl;
}
```

If the condition that stops the loop is compound (or always??), Use a Function for a Compound Boolean.

Extreme Values

You are writing code to find extreme values, for example the maximum and minimum in a collection or sequence of values.

What kind of loop do you use and how do you initialize the variables that track the extreme values?

You must process all items in the collection to ensure that the correct extreme values are found. The same forces are at play that help to identify a Definite Process All Items and a Iterator Process All Items loop.

The principle difficulty in writing code to find extreme values is determining initial values for the variables that represent the *extreme value so far* e.g., `currentMin` and `currentMax`. The invariant for `currentMin` is that it represents the minimal value of all the values processed so far. The best method for establishing this invariant is to use the first value of the collection for the initialization, but this can lead to the kind of duplicate code seen in the Loop and a Half pattern.

The range of values from which the extreme values are chosen is a force that affects the algorithm/code. For example, if extreme values are chosen from integers or doubles (floating point numbers) there are largest and smallest values; effectively there are values for infinity and negative infinity. For other types there may be no effective values for infinity, e.g., in choosing the largest or smallest string lexicogaphically there is no maximal string value (there may be a smallest string, usually the empty string `""` is less than any non-empty string).

Therefore, use the appropriate Process All Items loop and initialize extreme values to the first value in the collection from which extreme values are determined (care must be taken if the collection is empty). If possible, use indices, pointers, or references rather than values of objects/variables, i.e., keep track of the index of the current minimum in an array rather than the current minimal value. The value can be determined from the index, pointer, or reference.

Examples

Find the largest and smallest values in a vector of strings in C++. Use a Definite Process All Items loop and initialize current min and max indexing variables to the first value in the vector.

```
    vector<string> a;

    // code that puts values in a using push_back

    int minIndex = 0;    // index of minimum values between 0 and k
    int maxIndex = 0;    // index of maximum values between 0 and k

    int k;
    for(k=1; k < a.size(); k++)
    {
        if (a[k] < a[minIndex]) minIndex = k;
        if (a[k] > a[maxIndex]) maxIndex = k;
    }

    // minimal string stored in a[minIndex]
    // maximal string stored in a[maxIndex]
```

It's possible to write a generic function that returns the minimal value in a vector/array:

```
template <class Type>
Type min(const vector<Type> & a)
// precondition:  a contains a.size() values, values in a comparable by <
// postcondition: returns the index of the smallest value in a
{
    int minIndex = 0;   // index of minimal value in a[0..k]

    int k;
    for(k=1; k < a.size(); k++)
    {
        if (a[k] < a[minIndex]) minIndex = k;
    }
    return a[minIndex];
```

In Java this could be written as follows:

```
  class Extreme
  {
    /**
      * @param a is Vector of values that implement Comparable
      * @returns the the smallest object in the Vector
      */
    public static Object min(Vector a)
    {
        int minIndex = 0;
        int k;
        for(k=1; k < a.length(); k++)
        {
            if (a.elementAt(k).compareTo(a.elementAt(minIndex)) < 0)
            {
                minIndex = 0;
            }
        }
        return a.elementAt(minIndex);
    }
  };
```

You are finding the extreme values in a stream of string values using C++. The collection here uses an iterator interface, where the interface is the C++ convention of reading the stream and using the returned state of the stream to indicate when the iteration is finished.

The first values read, if any, are used to initialize the extreme values. It is not possible to use indexes or references to maintain the extreme values, so the values themselves are stored.

```
void findMinMax(istream & input, string & min, string & max)
// postcondition: min is the minimal value in stream input,
//                max is the maximal value in stream input
{
    string current;
    if (input >> current)
    {
        min = current;
        max = current;
    }

    while (input >> current)
    {
        if (current < min) min = current;
        if (current > max) max = current;
    }
}
```

There is duplicated code for reading values in the code fragment above. It's possible to avoid duplicated code when the range of values has effective values for positive and negative infinity.

You are finding the extreme values in a stream of double values. You want to avoid duplicated reading code. Therefore, initialize current min and max to positive and negative infinity (and use <= and >= for comparisons).

```
void findMinMax(istream & input, double & min, double & max)
// postcondition: min is the minimal value in stream input,
//                max is the maximal value in stream input
{
    double min = DBL_MAX;          // #include <float.h> for DBL_MAX
    double max = DBL_MIN;
    double current;

    while (input >> current)
    {
        if (current <= min) min = current;
        if (current >= max) max = current;
    }
}
```

One Loop for Linear Structures

You are writing Simply Understood Code(Gabriel), code whose intent and correctness are not apparent.

How do you write code that processes data stored in a linear structure, such as an array, list, or file?

Developing such code is not trivial, because you think that complex control structures are needed to implement the algorithm, or because of special cases.

Algorithmically, a problem may seem to call for multiple loops to match intuition on how control structures are used to program a solution to the problem, but the data is stored sequentially, e.g., in an array or a file. Programming based on control leads to more problems than programming based on structure.

Therefore, use the structure of the data to guide the programmed solution: one loop for sequential data with appropriately Guarded Conditionals(?) to implement the control.

The structure of the source of data is the guide for using this pattern, not the structure of how the data is processed or where information is written. For example, in processing a file representing a black-and-white bitmap stored as run-length encoded zeros and ones, e.g., 5 3 4 represents 000001110000, the structure of the file dictates using one loop while the structure of the image dictates using two loops. Use one loop because the data is stored in a file.

Examples

You are removing duplicated elements from a sorted array. At first you may think of writing nested loops: an outer loop to Process All Items and an inner loop to search past duplicated items (alternatively, search for a non-duplicated item). This approach can be a problem because a Definite Process All Items loop is called for, but the nested loop approach will make updating the indexing variable difficult. Instead, use only a Process All Items loop: one loop for linear structures:

```
void removedups(vector<string> & a, int & n)
// precondition: a[0] <= a[1] <= ... <= a[n - 1]
// postcondition: all duplicates removed from a,
//                n = # elements in a
{
    int k;
    int uniqueCount = 1;

    // invariant: no duplicates in a[0] .. a[uniqueCount-1]

    for(k=1; k < n; k++)
    {
        if (a[k] != a[uniqueCount-1])
        {
            a[uniqueCount] = a[k];
            uniqueCount++;
        }
    }
    n = uniqueCount;
}
```

A more elabore example of Quicksort Partition is given in an appendix.

Loop Invariant/Loop Initialization

You are writing a loop whose termination/continuation conditions are determined by values of local variables.

How do you determine the initial values of the variables?

The name of each variable should indicate its role in the loop and the invariant property that holds for the variable. A comment that expresses the invariant property should accompany each variable whose name does not immediately convey the invariant property.

Explicitly identifying the invariant for each variable used in the loop is a start. The invariant helps determine what the initial values are since the invariant must be true the first time the loop test is evaluated. See the example of removing duplicated elements in the One Loop for Linear Structures pattern.

For example, You are writing code to find the date on which Thanksgiving, the fourth Thursday in November, occurs. You decide to start on the first day of the month, and iterate through days counting Thursdays. Your first loop is:

```
Date d(11,1,1998);
int thursdayCount = 0;       // # of thursdays
while (thursdayCount < 4)
{
    if (d.DayName() == "Thursday")
    {
        thursdayCount++;
    }
    d++;
}

// d is Thanksgiving
```

You recognize a problem with this solution: `d` is incremented as the last statement in the loop so it will be one day after Thanksgiving when the loop terminates. You may decide to move `d++` before the if statement in the loop, or to rewrite the if statement as an if/else statement. The problem here is not with the loop body per se, it is with the initialization of the variable `thursdayCount`; what does `thursdayCount` count?

Therefore you must explicitly identify what is tracked/counted, what is the invariant property that holds for `thursdayCount`. In this case you want `thursdayCount` to be *the number of Thursdays in November on or before Date d*. You attempt to verify that the invariant holds the first time the loop test is evaluated and see that it will not hold when November 1 is a Thursday. You could initialize based on what day of the week November 1 falls on as shown below, but you want to avoid special cases.

```
 if (d.DayName() == "Thursday")
     thursdayCount = 1;
 else
     thursdayCount = 0;
```

To ensure that the invariant holds the first time the loop test is evaluated, you initialize `d` to the last day in October rather than the first day of November:

```
Date d(11,1,1998);
d--;                      // last day of October, month before
int thursdayCount = 0;    // # thursdays on or before d
while (thursdayCount < 4)
{
    d++;
    if (d.DayName() == "Thursday")
    {
        thursdayCount++;
    }
}

// d is Thanksgiving
```

External Patterns

We refer to each the following patterns in one or more patterns above. Some are patterns that we intend write in the future, and others are patterns documented by other authors. Links to on-line versions of the patterns are provided where available.

- Use a Function for a Compound Boolean
  - *Problem*: Even simple Boolean expressions can be hard to read. More complex Booleans can be downright intimidating. They interrupt the flow of code, and readers often cannot determine their meanings very easily.
  - *Solution*: Create a function that returns the value of the Boolean expression. Give the function an Intention Revealing Name.

- Simply Understood Code (Gabriel)
  - *Problem*: People need to be comfortable reading a piece of code before they feel confident that they understand it and can modify it.
  - *Solution*: "Arrange the important parts of the code so it fits on one page. Make that code understandable to a person reading it from top to bottom. Do not require the code to be repeatedly scanned in order to understand how data is used and how control moves about."

- Assign Variables Once (Gabriel)
  - *Problem*: If a variable is assigned twice, you might have trouble figuring out which assignment provides the value at a particular point, and your inclination is to remember the last assignment saw.
  - *Solution*: Assign local variables only once, and, if possible, do that at the place where the local variable is defined.

- Local Variables Reassigned Above Their Uses (Gabriel)
  - *Problem*: "Sometimes a piece of code needs to re-assign local variables more than once. If this is done without paying attention to a person reading the code who is unfamiliar with it, misunderstandings are easy."
  - *Solution*: "A local variable that must be re-assigned should be re-assigned in a place that is textually above where it is used or referenced."

- Intention Revealing Name (Called "Intention Revealing Selector" by Beck)
  - *Problem*: How do you name a procedure?
  - *Solution*: Name procedures for what they accomplish.

- Composed Procedure (Called "Composed Method" by Beck)
  - *Problem*: How should you divide a program into components?
  - *Solution*: "Divide your program into [procedures] that do one identifiable task. Keep all of the operations in a [procedure] at the same level of abstraction. This will naturally result in programs with many small [procedures], each a few lines long."

- Role Suggesting Temporary Variable Name (Beck)
  - *Problem*: What do you call a temporary variable?
  - *Solution*: "Name temporary variables for the role they play in the computation. Use variable naming as an opportunity to communicate valuable tactical information to future readers."

- Caching Temporary Variable (Beck)
  - *Problem*: How do you improve the performance of a method, in the face of repeated

    expression evaluations?
    - *Solution*: "Set a temporary variable to the value of the expression as soon as it is valid. Use the variable instead of the expression in the remainder of the method."

- An Alternative Action selects from among two or more actions on the basis of some condition.

Acknowledgements

We would like to thank the organizers of ChiliPLoP'98 for the opportunity to gather and to begin to form an elementary patterns community. We thank especially the other members of our ChiliPLoP workshop for their thoughts on looping patterns: Joe Bergin, Robert Duvall, Ed Epp, and Rick Mercer. Our PLoP shepherd, Marc Bradac, made many suggestions that improved our initial efforts.

References

1. Owen Astrachan. Design Patterns: An Essential Component of CS Curricula. *SIGCSE Bulletin and Proceedings*, Volume 30(1):153-160, March 1998.

2. Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, New York, 1997.

3. Jon L. Bentley and M. Douglas McIlroy. Engineering a Sort Function. *Software Practice and Experience*. Volume 23(11):1249-1265, November 1993.

4. James O. Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.

5. Richard Gabriel, Simply Understood Code, `http://c2.com/cgi/wiki?SimplyUnderstoodCode`

6. Eric S. Roberts. Loop Exits and Structured Programming: Re-opening the Debate. *SIGCSE Bulletin and Proceedings*, Volume 27(1):268-272, March 1995.

7. Sartaj Sahni. *Data Structures, Algorithms, and Applications in C++*, McGraw-Hill, 1997.

8. Eugene Wallingford Roundabout, A Pattern Language for Recursive Programming, PLoP-97, `http://www.cs.uni.edu/~wallingf/research/patterns/recursion.html`

Appendix: The Quicksort Partition

This is a more elaborate example of One Loop for Linear Structures.

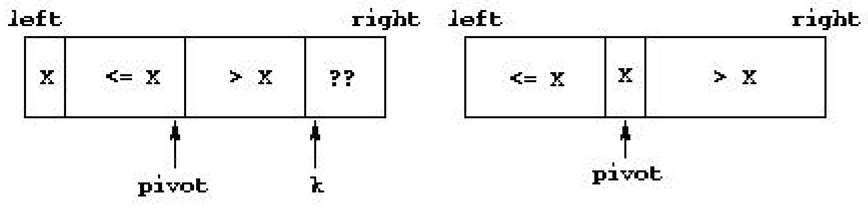
The partition phase of quicksort for arrays is a classic example. Most textbooks use a complicated solution in which an outer loop is essentially a Process All Items loop wrapped around two inner loops that move indices delineating the two sections of the array. This three-loop solution is difficult for students to understand, easy to code incorrectly, and harder to generalize to a three-phase partition required to avoid bad asymptotic performance when duplicate elements are stored in an array.

Using One Loop for Linear Structures in conjunction with Process All Items leads to the code shown below (taken from Engineering a Sort Function).

```
int partition(vector<string> & a, int left, int right)
// precondition:  left <= right
// postcondition: rearranges entries in vector a
//                returns pivot such that
//                forall k, left <= k <= pivot, a[k] <= a[pivot] and
//                forall k, pivot < k <= right, a[pivot] < a[k]
//
{
    int k, pivot = left;
    string center = a[left];

    for(k=left+1, k <= right; k++)
    {
        if (a[k] <= center)
        {
            pivot++;
            swap(a[k],a[pivot]);
        }
    }
    swap(a[left],a[pivot]);
    return pivot;
}
```

The invariant for this can be shown pictorially.



Not only is this code simpler to understand than the code shown in many texts, it is more easily generalized to the three-phase "fat partition" (see Bentley) in which elements equal to the pivot are treated separately and put in their own partition. A fat partition scheme is necessary to avoid *$O(n^2)$* performance for arrays with many duplicate items. Fat partitioning yields a section of elements equal to the pivot element as well as sections less than and greater than the pivot element (two values are returned from the function and no recursion occurs on the equal section). It is also simple to include a median-of-three partition by swapping the median value into the left-most entry before the loop begins.

The code below is indicative of partition functions developed without the benefit of the one loop for linear structures pattern, it appears in the text by Sahni. On the surface this code is harder to reason about and harder to adapt to the "fat partition" scheme.

```
int partition(vector<string> & a, int left, int right)
// precondition:  left <= right
// postcondition: rearranges entries in vector a
//                returns pivot such that
//                forall k, left <= k <= pivot, a[k] <= a[pivot] and
//                forall k, pivot < k <= right, a[pivot] < a[k]
//
{
    int i = left,
        j = right+1;
    string pivot = a[left];

    while (true)
    {
        do{
            i++;
        } while ( a[i] < pivot);

        do{
            j--;
        } while (a[j] > pivot);

        if (i >= j) break;

        swap(a[i],a[j]);
    }
    a[left] = a[j];
    a[j] = pivot;
    return j;
}
```

Code developed with multiple loops can be more efficient. For example, the final, fully-tuned version of partition in Bentley uses the loops-in-a-loop coding style shown in the Sahni example.