Some Patterns for CS1 Students*

Berna L. Massingill†

Abstract

Students in beginning programming courses struggle with many aspects of programming. This paper outlines some patterns intended to be useful to these beginners; they represent advice I have frequently offered (or wanted to offer) to students in beginning courses.

1 Introduction

This paper outlines some patterns intended to be useful to students in beginning programming courses; they represent advice I have frequently offered (or wanted to offer) to students in beginning courses. They are intended to be read by the students, and possibly their instructors as well. The patterns, with their problem statements, are as follows.

***Comments as Outline*.** You have been given a problem and asked to produce both code to solve it and appropriate documentation. Is there some way to meet both goals together?

***Pictures of Data Structures*.** You need to define or manipulate a data structure and do not immediately see how to express the ideas in pseudocode.

***Variables as Invariants*.** How can you define and use variables in a way that makes it easier to produce working code?

***Code Others Can Read*.** How do you write code in a way that makes it readable to others, particularly people from whom you might want assistance?

†Department of Computer Science, Trinity University, San Antonio, TX; bmassing@trinity.edu.

2 *Comments as Outline*

Problem

You have been given a problem and asked to produce both code to solve it and appropriate documentation. Is there some way to meet both goals together?

Context

Most instructors ask that students turn in code that not only does what "it's supposed to" but includes comments describing what it does.

Forces

- Writing comments can seem like an annoying distraction from the job of producing working code.
- Code without comments, however, is more difficult for human readers to understand — including the code's author a year down the road.
- Further, it is difficult to solve a problem if you don't fully understand it, and diving too quickly into details of programming syntax is often not the shortcut to working code that it might appear to be.

Solution

Generations of students have been taught to approach the job of writing an essay by first producing an outline, the idea being that this encourages the student to think through the overall structure of the essay and what points it should make before diving into the production of prose. (Many students resist this approach. It's still a good idea, and worth trying a few times to see whether it helps.) A similar approach can be taken in programming by first writing the comments (the "outline") and then writing the code (the essay). For most programs it will probably make sense to start with a broad outline (the program's documentation) and then add levels of detail (subprogram documentation and then pseudocode), as follows.

Documentation. Before writing a program or subprogram, write its documentation. Documentation has two potential roles: describing (for the benefit of potential users) what the program does, and describing (for the benefit of readers of the actual code) anything tricky or interesting about how it works. Which one of these you should emphasize depends on who will read your documentation.

Describing what the program does. This aspect of the documentation usually includes the following.

- A general short description of what it is supposed to do (focusing on "what" rather than "how" — e.g., "compute and print the variance of *N* doubles read from standard input" rather than "read *N* doubles into an array" etc.).

- A description of its inputs and outputs. Inputs include input files, "standard input", input from a keyboard/mouse, and parameters passed to subprograms, and values of global variables. Outputs include output files, "standard output", output to a screen, parameters passed to a subprogram and changed, return values, and changed values of global variables.
- What the program expects from its inputs. (For example, a function to compute and return real-valued square roots might expect its input to be non-negative.)
- How the program's outputs depend on its inputs. (For example, if input to a square-root function is a non-negative double x, the function's output is a non-negative double y such that $y^2 = x$.

Writing such comments before starting to write code forces you to think more carefully about the problem before diving into syntactic details. It also greatly improves the odds that the comments will actually get written.

Describing how the program works. As a philosophical position, "the documentation should describe what the program does, not how it does it" is very attractive. However, if other developers will read your code, and there are things about how it works that won't be obvious from reading the code itself, it can be helpful to add a block of comments to the top of the program describing anything interesting or tricky or otherwise noteworthy.

Pseudocode. Before starting to write detailed code for a (sub)program, write a pseudocode version that represents your overall logic. This pseudocode then becomes the subprogram's internal documentation.

Also recognize that this may be an iterative process; as you write the pseudocode you may realize that you've neglected to think through all the cases. If that happens, go back and update the "documentation" part of your outline as well, as in the first example below. You may also recognize a need for additional subprograms, which you can then add to your overall outline, first writing their documentation.

Examples

(Examples at present are limited to Java, but the ideas apply to any language that allows comments.)

Program to compute averages, version 1

For example, consider a simple program to compute the average of a set of numbers. The program's inputs are command-line arguments, which should be numbers. Its output should be the average of these numbers. Fig. 1 shows a first pass at writing its documentation. We'll go ahead and include the method signature for the main method, since we know what that should look like. (`@param` is a tag for the `javadoc` tool.) Notice that these comments don't say anything about *how* we're going to compute the

```
/**
 * Main method:
 *
 * Program input:
 *    Zero or more numbers (n1, n2, ...), passed as command-line
 *    arguments.
 * Program output:
 *    Average of n1, n2, ...., written to standard output.
 *
 * @param args command-line arguments -- numbers to average
 */
public static void main(String[] args) { }
```

Figure 1: First step in outlining program to compute average from command-line arguments.

average; they just describe the program's inputs and outputs. Someone using the program presumably doesn't really care how it arrives at the answer, provided the answer is correct.

Now we can continue outlining the program at a lower level of detail, that is, by writing pseudocode, as shown in Fig. 2. Here the overall strategy is reasonably obvious: What we have is an array of `String` objects (and we know how many of them we have); to compute the average of the numbers they represent, we should turn them all into numbers, add them up, and divide the sum by the number of arguments. The

```
/**
 * Main method.
 *
 * Program input:
 *    Zero or more numbers (n1, n2, ...), passed as command-line
 *    arguments.
 * Program output:
 *    Average of n1, n2, ...., written to standard output.
 *
 * @param args command-line arguments -- numbers to average
 */
public static void main(String[] args) {
    // initialize sum to zero
    // for each command-line argument, convert to number and
    //   add to sum
    // compute average (divide sum by args.length) and print
}
```

Figure 2: Second step in outlining program to compute average from command-line arguments.

idea is to write pseudocode that's detailed enough that turning each line into actual code is pretty straightforward. (In terms of the "writing an essay" analogy, we're now at the point where we have a complete outline and can turn each outline point into a sentence or sentences, which we should be able to do without thinking too much about

the overall structure.) In this example, thinking a little about whether this pseudocode is detailed enough reveals some problems. First, the pseudocode "convert to number" seems easy enough to turn into code (assuming we know about the appropriate Java library method), but what if the argument is a string that doesn't represent a number ("hello", for example)? Second, we plan to compute the average by dividing a sum by `args.length`, but what if `args.length` is zero? Maybe we should have thought of both of these problems earlier, but if we didn't, we can consider them now and decide what we should do if the input isn't what we expect. A reasonable choice is to have the program recognize bad input (no arguments, or a non-numeric argument) and print some sort of error message. We could add this to the program-in-work as shown in Fig. 3. (In real-world programs it might be overkill to describe in the program documentation all the ways in which input could be "bad" and what the program does if it is, but we'll go ahead and do it for the sake of completeness. We can now produce

```
/**
 * Main method.
 *
 * Program input:
 *    One or more numbers (n1, n2, ...), passed as command-line
 *    arguments.
 * Program output:
 *    Average of n1, n2, ...., written to standard output, or an
 *    error message if there are no arguments or one of the
 *    arguments was not a number.
 *
 * @param args command-line arguments -- numbers to average
 */
public static void main(String[] args) {
    // if no arguments, print error message and exit
    // initialize sum to zero
    // for each command-line argument, convert to number and
    //   add to sum (if unable to convert, print error message
    //   and exit)
    // compute average (divide sum by args.length) and print
}
```

Figure 3: Second step in outlining program to compute average from command-line arguments, revised.

the actual program, treating each line of pseudocode as an "outline point" to be fleshed out, as shown in Fig. 4.

Program to compute averages, version 2

Now suppose we want to revise the program so that it gets its input from standard input rather than from command-line arguments. To simplify the logic, we'll make the interface Unix-like: We won't prompt the user, and we'll read input until we detect "end of file".[1] As with the first version of the program, it's easy to imagine situations in

[1] "End of file" is signaled interactively by a system-dependent key sequence —control-D on Unix systems.

```
/**
 * Main method.
 *
 * Program input:
 *    One or more numbers (n1, n2, ...), passed as command-line
 *    arguments.
 * Program output:
 *    Average of n1, n2, ...., written to standard output, or an
 *    error message if there are no arguments or one of the
 *    arguments was not a number.
 *
 * @param args command-line arguments -- numbers to average
 */
public static void main(String[] args) {
    // if no arguments, print error message and exit
    if (args.length <= 0) {
        System.out.println("Arguments are numbers to average");
        System.exit(1);
    }
    // initialize sum to zero
    double sum = 0.0;
    // for each command-line argument, convert to number and
    //   add to sum (if unable to convert, print error message
    //   and exit)
    for (int i = 0; i < args.length; ++i) {
        try {
            sum += Double.parseDouble(args[i]);
        }
        catch (NumberFormatException e) {
            System.out.println("Unable to convert argument '" +
                    args[i] + "' to a number");
            System.exit(1);
        }
    }
    // compute average (divide sum by args.length) and print
    System.out.println("Average = " + sum/args.length);
}
```

Figure 4: Program to compute average from command-line arguments.

which the input would be "bad" (no input, or input that's not numeric), so we'll take that into account from the start. Fig. 5 shows a first pass at writing documentation. Notice

```
/**
 * Main method.
 *
 * Program input:
 *    One or more numbers (n1, n2, ...) read from standard input.
 * Program output:
 *    Average of n1, n2, ...., written to standard output, or
 *    an error message if there is no input or one of the inputs
 *    was not a number.
 *
 * @param args command-line arguments -- not used
 */
public static void main(String[] args) { }
```

Figure 5: First step in outlining program to compute average from numbers read from standard input.

that again these comments don't say anything about *how* we're going to compute the average (do we read the input into an array, or can we process each line as we read it, or what?); they just describe the program's inputs and outputs. Again, the objective is to document what's of interest to someone using the program — what input the program expects and what output it produces.

Now we can continue outlining the program at a lower level of detail, that is, by writing pseudocode, as shown in Fig. 6. The overall strategy isn't quite as obvious as in the previous example, but it's still not too complicated: We'll keep two "running totals" (a sum and a count of lines read). We'll keep reading lines from standard input until there are no more to be read; as we read each line, we'll increment the count of lines read and the sum. Notice the initial comments describing the overall strategy. We can now produce the actual program, treating each line of pseudocode as an "outline point" to be fleshed out, as shown in Fig. 7. Notice that in doing this we made a small change to the pseudocode: We collapsed the two lines "while there's input to read" and "read a line" into one line of code, since the simplest way to determine whether there's input to be read is to attempt to read it.[2]

Related Patterns

Before writing pseudocode it may be helpful to use the *Pictures of Data Structures* and *Variables as Invariants* patterns to clarify your thinking about a program's logic.

[2]In turning the pseudocode into code, we also have to deal with the fact that almost all of the Java I/O library methods are declared to throw `IOException`s, which must be explicitly dealt with. For this program dealing with these exceptions is more a matter of keeping the compiler happy than dealing with potential run-time errors —it's hard to imagine circumstances in which reading from standard input would throw one of these exceptions —so we don't explicitly address this issue in the pseudocode.

```
/**
 * Main method.
 *
 * Program input:
 *    One or more numbers (n1, n2, ...) read from standard input.
 * Program output:
 *    Average of n1, n2, ...., written to standard output, or
 *    an error message if there is no input or one of the inputs
 *    was not a number.
 *
 * @param args command-line arguments -- not used
 */
public static void main(String[] args) {

    // overall strategy uses two running-total variables,
    //   "sum" (of values read so far) and "count" (of lines
    //   read so far)

    // initialize sum and count to zero
    // set up to read from standard input a line at a time

    // while there's input to read:
    //   read a line
    //   increment count of lines read
    //   convert line to number and add to running total (if
    //     unable to convert, print error message and exit)

    // if at least one input, compute average (divide sum by
    //   args.length) and print
    // else print error message
}
```

Figure 6: Second step in outlining program to compute average from numbers read from standard input.

```
/**
 * Main method.
 *
 * Program input:
 *    One or more numbers (n1, n2, ...) read from standard input.
 * Program output:
 *    Average of n1, n2, ...., written to standard output, or
 *    an error message if there is no input or one of the inputs
 *    was not a number.
 *
 * @param args command-line arguments -- not used
 */
public static void main(String[] args) {

    // overall strategy uses two running-total variables,
    //   "sum" (of values read so far) and "count" (of lines
    //   read so far)

    try {

        // initialize sum and count to zero
        double sum = 0.0;
        int count = 0;

        // set up to read from standard input a line at a time
        BufferedReader rdr =
            new BufferedReader(new InputStreamReader(System.in));
        String line = null;
        // while there's input to read (test for input also reads
        //   it):
        while ((line = rdr.readLine()) != null) {
            // increment count of lines read
            ++count;
            // convert line to number and add to running total
            //   (if unable to convert, print error message and
            //   exit)
            try {
                sum += Double.parseDouble(line);
            }
            catch (NumberFormatException e) {
                System.out.println("Unable to convert input '"
                        + line + "' to a number");
                System.exit(1);
            }
        }
        // if at least one input, compute average (divide sum
        //   by count) and print
        if (count > 0) {
            System.out.println("Average = " + sum/count);
        }
        // else print error message
        else {
            System.out.println("Error -- no input");
            System.exit(1);
        }

    }
    catch (IOException e) {
        // should never happen!
        e.printStackTrace();
        System.exit(1);
    }
}
```

Figure 7: Program to compute average from numbers read from standard input.

3 *Pictures of Data Structures*

Problem

You need to define or manipulate a data structure and do not immediately see how to express the ideas in pseudocode.

Context

Most programs involve the use of some type of data structure. Many assignments in programming courses are specifically aimed at teaching students how to define and manipulate well-known data structures. How these data structures "work" is often difficult to understand from the code that implements them.

Forces

- Anything that doesn't directly produce compilable code can seem like an annoying distraction.
- However, it is difficult to write correct code for data structures such as linked lists and trees without some sort of mental image of how the parts fit together.

Solution

Draw pictures of the data structures involved, and work through examples of the operations to be performed using the pictures. Almost any explanation of a nontrivial data structure will involve diagrams that should help the reader understand how it is put together and how the operations on it work. There's a reason for the ubiquity of these diagrams — they're an excellent way to develop a mental model of what's supposed to be going on, and once you have this mental model it's usually much easier to write the code.

Examples

Linked lists

The standard picture of a linked list looks something like the one in Fig. 8. If you

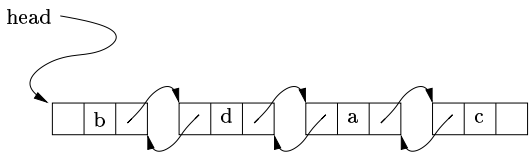

Figure 8: Doubly-linked list of characters.

were going to write code that "walks through" the list and prints the value stored at

each node, you would likely want some sort of pointer/reference that first points to the first node, then to the next one, and so forth. You could represent this pictorially with a picture such as the one in Fig. 9. You could imagine (or draw) the "current node"

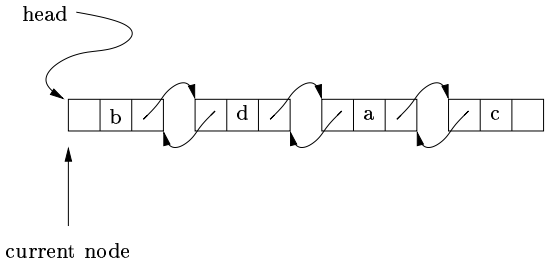

Figure 9: Doubly-linked list of characters with "current node" pointer.

pointer moving through the list to point to the second and subsequent nodes.

Pictures would be even more helpful before starting to write code to add or remove elements. For example, if you want to write code to find and remove an element given its (character) value, you might start with the picture in Fig. 8 and consider deleting the first element in the list (`b`), the last element in the list (`c`), an element in the middle (`d` or `a`), and an element not in the list at all (`x`, for example). For each case, first figure out what the updated picture should look like (for example, if you delete `b` you must change the `head` pointer; for other examples you will probably have to change some of the node-to-node pointers), and then figure out what your code has to do to accomplish this. Pictures such as the ones in Fig. 9 can be helpful in figuring out how to accomplish a given change.

If the elements in the list are supposed to be "in order" (as determined by the values stored), it is easy to incorporate this into the pictures.

Other examples

Pictures are probably helpful in dealing with almost any nontrivial data structure. Other examples used in introductory classes are stacks (implemented using arrays or linked lists), queues (implemented using arrays or linked lists — and pictures may be particularly helpful for a "circular queue" implemented using an array), and trees (including sorted binary trees and heaps).

Related Patterns

In moving from pictures to code, it can be helpful to use the *Variables as Invariants* pattern.

4 *Variables as Invariants*

Problem

How can you define and use variables in a way that makes it easier to produce working code?

Context

You understand the problem (from using the *Comments as Outline* and *Pictures for Data Structures* patterns, perhaps). Now you need to define variables and write (pseudo)code using them.

Forces

- Writing code that works without extensive trial-and-error debugging can be difficult.
- Theoretical work on formal proofs of program correctness defines a notion of "invariants" that encourages viewing programs in terms of "things that are always true" rather than in terms of explicitly thinking about all the paths one could take through a large block of code. Formal correctness arguments are overkill for many situations, but adopting the underlying way of thinking about programs can make reasoning (informally) about program correctness more tractable.

Solution

Every time you declare a local variable (except for trivial things such as loop counters), you should have a clear idea of what the variable is supposed to represent. (Ideally this idea should be apparent from the variable's name; if not, add a comment to the declaration that makes it clearer.) Then write the code to be consistent with this idea of what the variable is supposed to represent. This is difficult to explain in general terms, but by considering some examples (in the next section) the notion should become clearer.

Examples

Stack implemented using an array

For example, suppose you are implementing a stack data structure using an array. You need to keep track of where the topmost element of the stack resides. One way is to define a variable `top` that represents the index of the topmost element. This variable needs to be changed any time an element is pushed onto or popped from the stack. You can also determine whether the stack is full or empty using this variable — if it's the same as the last index in the array, the stack is full, and if it's one less than the first index in the array, the stack is empty. An alternative that avoids negative values (for empty stacks) is to define a variable `nextFreeSpot` that represents the index of the element just above the top of the stack.

Binary search

As another example, suppose you are implementing binary search in an array. Useful variables might be ones representing the first and last indices to search (`firstToSearch` and `lastToSearch`, say). Every time you compare an element of the array to the element being searched for, you change one of these indices; when `firstToSearch` becomes greater than `lastToSearch`, the search stops.

Reading and processing input

As another example, consider the second version of the "compute an average" program in the *Comments as Outline* pattern (Sec. 2). The variable `count` represents a count of lines read so far; the variable `sum` represents the sum of the values found on those lines (one per line). Initially (before any lines are read) both are 0. Every time we read a line from the input, both of these variables are updated to reflect the new state of affairs (one more line has been read). When all the input has been read, we know that `count` is the number of values we've read and `sum` is their sum, and it's easy to see what we have to do to compute the desired average.

Processing linked lists

As another example, consider "walking through" a linked list, as discussed in the *Pictures as Data Structures* pattern (Sec. 3). In the picture, there's a pointer to a "current node" that moves as you "walk through" the list. To turn this picture into code, it makes sense to define a variable `currentNode` that starts by pointing to the first node in the list and then moves through the other nodes, following their "next" pointers. If you think of this variable as "something that points to the first node not yet visited (printed)", you can then write down pseudocode as shown in Fig. 10. On each trip through the loop, `currentNode` points to the first node not yet printed/visited, so we first print the associated value and then advance `currentNode` to the next node; at the end of the list, `currentNode` becomes null and all nodes have been visited.

```
// initialize currentNode to head
// while currentNode is not null:
//   print value for node pointed to by currentNode
//   advance currentNode to point to next node
```

Figure 10: Pseudocode to "walk through" a list whose first node is `head` and print the value in each node.

Related Patterns

This pattern can be helpful in translating this pseudocode of the *Comments as Outline* pattern and the pictures of the *Variables as Invariants* pattern into code.

(The name of this pattern comes from the notion of "loop invariants" as presented in the context of formal proofs of program correctness and from similar notions of "invariants" in discussions about correctness of concurrent algorithms.) Briefly, in that context, a loop invariant is a statement about program variables that is true before the loop starts and is "preserved" by each loop iteration (if it's true before the iteration starts, it's true after it ends). By combining a well-chosen loop invariant and information about when a loop ends, you can reason about what has to be true when the loop ends.

5 *Code Others Can Read*

Problem

How do you write code in a way that makes it readable to others, particularly people from whom you might want assistance?

Context

Many syntactic details of how code is arranged are of no importance in determining the program's meaning (e.g., most whitespace in a Java program could be omitted without changing the program's meaning) but can greatly affect how easily human readers can grasp its meaning. Programming instructors are apt to nag students about things that don't affect the code's meaning, only its readability — indenting code in a consistent way, following language conventions for variables names, etc.

Forces

- Writing code that conforms to prevailing standards of naming and layout can seem pointless and unduly confining.
- Reading code that ignores prevailing standards of naming and layout is more difficult. To some extent, the more experienced the reader is with a language, the more he/she is apt to "expect" code to follow prevailing standards and to be distracted by departures from these standards.

Solution

Make it easier for experienced programmers to read your code by conforming to their standards of naming and layout. This makes it easier for them to help you locate errors. The "principle of least surprise" applies here: Surprising the reader with nonstandard usage or hard-to-follow layout distracts him/her from the more important things (program logic, language syntax/semantics) with which you might want help.

Examples

Java naming conventions

Most experienced Java programmers adhere to the following conventions when naming classes and variables:

- Names of classes and interfaces start with a capital letter — e.g., `System`, `Math`, or `MyObject`. If the name is logically more than one word, all words are capitalized — the so-called "camel case" style (e.g. `BufferedReader` or `ColorChooserModel`).
- Names of "constants" (static final variables) are in all caps — e.g., `Math.PI` or `Color.RED`. If the name is logically than one word, underscores are used to separate words (e.g., `Color.DARK_GRAY`).

- Names of methods and variables start with a lower-case letter — e.g., `readLine` or `out`. If the name is logically more than one word, all words except the first one are capitalized, similar to the usage for classes and interfaces.

Thus, an experienced Java programmer expects `MyThing` to be a class or interface and `myOtherThing` to be a variable or method and is distracted by code that uses the opposite convention, or no convention at all.

Program indentation

There are many different styles of program indentation, and in some circles you can start a quasi-religious war by asking which one is best. The more tolerant will say "just use some consistent scheme that more or less reflects the structure of the program." Some different styles of presenting a simple loop are shown in Fig. 11. (Guess which one is *not* recommended.)

```
// One indentation style.
for (int i = 0; i < 100; ++i) {
    for (int j = 0; j < i) {
        System.out.print("*");
    }
    System.out.println();

// Another indentation style.
for (int i = 0; i < 100; ++i)
  {
  for (int j = 0; j < i)
    {
      System.out.print("*");
    }
  System.out.println();
}

// Yet another indentation style.
for (int i = 0; i < 100; ++i) {
for (int j = 0; j < i) {
System.out.print("*"); }
System.out.println(); }
```

Figure 11: Nested `for` loops, different indentation styles. (Not all are recommended.)

Acknowledgments

I gratefully acknowledge the help of my shepherd for this paper, Neil Harrison.