On Design and Design Documents

Norman Ramsey

Fall 2008

Introduction

Engineering is design under constraint. Sometimes a constraint is imposed from outside (data must fit on the computer's hard drive) and sometimes a constraint embodies what we want from a system (I want a backup server that draws at most 10 watts of power, so I can leave it on all the time.)

In computer systems, *design* means breaking a system into pieces which cooperate to produce a harmonious, useful whole. For the software parts of a design, the best tool we have for understanding what a piece is and how it cooperates with other pieces is the *interface*.

When I give a programming assignment, I often ask for a *design document*. A design document is one of many ways to describe software, and software engineers have a huge vocabulary for talking about descriptions. I don't worry about the vocabulary, but an analogy with homebuilding might be helpful. If I want a 4-bedroom Colonial, that might be called a *requirement*; if I know how many square feet and of what materials, that might be part of the *specification*; brief descriptions of plan and elevation might constitute a *design*; blueprints or architectural drawings might be *detailed design*, and of course the house itself is the *implementation*.

In class, I ask for a design document to increase your chances of producing a correct and complete implementation without pain. Writing a design document develops your ability to think about your design, and such thought will reduce the number of costly and unplanned changes to your software program (i.e., late nights). The design document helps you think before you code.

Alert!

Design documents are due well before implementations. Take proper advantage of this structure: start thinking about how to design your program before you start writing it. Think first, code later, finish early, and enjoy life.

Contents of a Design Document

To help your reader (and yourself) focus on whatever part of your design may be relevant at any given time, I ask that you break down your design document into clearly labelled sections. You'll want to write these sections (in the order in which they appear):

- *Problem statement.* In one or two sentences, state the problem that your program will solve. Make it short and high-level: how would you tell your roommate?

  (This section helps verify that you have understood the problem.)

- *Use cases.* In a few sentences, say how your program will be used. Describe who or what will interact with your program: a person? Another program? Most important, *show an example* of how your program will be used, perhaps giving sample input and output.

  (This section helps solidify your understanding of the problem. It may also suggest a plan for testing your code.)

- *Assumptions and constraints.* Do you need to make assumptions about what sorts of input you expect—where input comes from, what happens after the program executes, or other constraint?

  (This section helps bring focus to open-ended problems. For simple, clearly stated problems it is of minimal use and should be short.)

- *Architecture.* What are the major components in your system and what are their *interfaces*? How do components interact? From least formal to most formal, you might

  – Draw pictures
  – Sketch the contents of each interface using some sort of pseudocode
  – Write out interfaces in a suitable programming language

  Pseudocode is often a good path.

  (*This section is the most important section* and will take you the most time to write. It will help you develop a full understanding of the assignment and will also require serious thinking about how you plan to attack the problem.)

- *Implementation plan.* What algorithms and/or *data structures* will each component need? Which pieces of your program will you build first? What will you build yourself? What will you reuse from a library? How long will each component take you? When will it be completed? Give dates and times![1]

[1] It may seem very strange to give dates and times in an implementation plan, especially when that plan may not survive first contact with the code. But in the real world, you will be asked over and over to estimate how long your work will take. I'm giving you a chance to practice this skill now, when the stakes are low.

The most critical aspect of your implementation plan is to decide on the *representation* of your abstract data types. Paraphrasing what Fred Brooks wrote in *The Mythical Man-Month*,

> Representation is the essence of programming... Much more often, strategic breakthrough will come from redoing the representation of the data. This is where the heart of a program lies. Show me your procedure bodies and conceal your data-type declarations, and I shall continue to be mystified. Show me your data-type declarations, and I won't usually need the bodies of your procedures; they will be obvious.

(This section helps make sure that your architecture is actually implementable, or that if it isn't, you'll discover it quickly. Moreover, it increases the chances that writing your program will bring you joy, not drudgery, confusion, or loss of sleep.)

- *Test plan.* How will you convince anybody (starting with yourself) that your program works? What test cases will you use? Be specific. Give examples of test cases. Plan for corner cases and error conditions. *You will often need to write test code that does not have to be turned in as part of the assignment.*

  (This section takes your high-level ideas about how your program will be used and force you to come up with concrete ways of testing to see if your program actually meets requirements.)

Advice on your implementation plan

The key to successful implementation is to *get an end-to-end solution working as quickly as possible.* You want a program that does *something* which you can work with and then improve. Programming is easier and more fun when your code always does something.

Something trivial is a good place to start. For example, if your ultimate goal is to read a bitmap, remove black edges, and write a modified bitmap, you might have a plan like this:

1. Read a bitmap and write the same bitmap on standard output, without ever putting a pixel into a data structure. Make a quick check for correctness using an image viewer (15 minutes).
2. Read a bitmap into a two-dimensional array of bits, then write the bitmap to standard output in PBM format (30 minutes).
3. Perform a trivial transformation on the bitmap, like changing every black pixel to white and every white pixel to black. Check in image viewer (10 minutes).
4. Identify black edge pixels and make them white (30 minutes).

After the design document

Once you've finished your design, start with your implementation plan. As you proceed, you may find use cases, requirements, and corner cases that you did not anticipate. In response, your design might change. It's all good—my goal is to have you think about design decisions; you won't usually need to update your design document to reflect new decisions. If I ever want an updated design document, I'll let you know.

When you have difficulty

Both design and programming will challenge your intellect and ingenuity. Don't let anybody kid you that it's easy,[2] and if you don't know how to get started, don't get discouraged. One benefit of the design document is it's a cheap way of discovering that you're confused. Take whatever you've written and bring it to a member of your course staff. Everybody on the staff likes teaching, and we'll do our best to ask you questions that will get you on track. Eventually you'll learn to ask yourself helpful questions!

Frequently asked questions

Why do I have to write a design document? Computer scientists need an education in design as well as programming. Design is one of the most interesting, challenging, and fun things computer scientists get to do. When you graduate, I want you never to be stuck in a situation where other people have the fun of designing the programs you write.

How long and how much detail should be in my design document? Regarding detail, focus on the architecture, especially its novel features. Detailed code is usually out of place: the design document focuses on the *interaction* of components, not the *implementation* of each component. Your implementation plan should include more detail than just what components will be written in what order. *When* will you write which components? *How long* will they take? *How will you test* your system when it is still small and manageable, but incomplete? Regarding length, provided you give the information needed, shorter is always better. Overall, your design document should embody a clear, concise, and simple specification for the implementation you plan.

An obsessive-compulsive design document

Here's an acceptable design document. This one is over the top; there's much more detail than necessary. The extra detail is there to give you ideas about

[2]Part of the culture of computer science is that when working engineers are collaborating on a design, someone may say, "from this point on it's just a Simple Matter of Programming." The secret passphrase is uttered by one programmer to another when both parties know well that almost nothing about programming is simple. Properly translated, the statement means "well, we've done a lot of preliminary work, and things look good enough that now we can get started on the hard stuff."

how to make an implementation plan in a way that gives you an end-to-end system as soon as possible—and that lets you test as you go.

- *Problem.* Implement a sort function that can be used by C programmers to sort lists of double-precision floating-point numbers.
- *Use case.* Here's a use case that sorts a statically initialized array in place.

```
/* user defines a list of doubles */
double magic_numbers[] = { 0.0, 3.1415926535, 2.718281816, 1.0, 1e-179 };

/* user defines comparator to sort in ascending order */
int compare_double(double x, double y) {
  return x == y ? 0 : x < y ? -1 : 1;
}

/* we use a macro good for any statically initialized array in C */
#define NELEMS(A) (sizeof(A)/sizeof((A)[0]))

/* we sort the array in place */
void sort_magic() {
  sort(compare_double, magic_numbers);
    /* now as postcondition, the list magic_numbers is sorted in place */
}
```

- *Assumptions and constraints* The sort function is to be provided with a pointer to the first element of the array to be sorted, the length of the array, and a comparison function that determines the sort order.
  - The pointer to the array is non-NULL.
  - The array is an array of `doubles`.
  - The comparison function is a valid function pointer; otherwise we have an unchecked run-time error.[3]
  - The comparison function must provide a total ordering; to provide a comparison function that is not a total ordering is an unchecked run-time error.
  - The comparison function should take two inputs x and y. If x must precede y in the sorted list, the comparison function should return -1. If x must follow y in the sorted list, the comparison function should return 1. If x and y can appear in either order (because they are considered equal), the comparison function should return 0.
  - The sort mutates the input list in place, so it is a destructive sort.

[3]It is an annoying defect in the C language standard that there is no portable way to compare a function pointer and a null pointer; on some architectures, comparing a function pointer with `NULL` will not even compile. For example, on some older machines a data pointer (like `NULL`) is a 16-bit value but a function pointer is a 32-bit value.

  - The mutated list will be totally ordered according to the comparison function. More precisely, for any suitable i, `compare(a[i], a[i+])` may be -1 or 0, but it will never be 1.
  - In the worst case, the sort algorithm should require $O(n\ log(n))$ space and time.
  - The sort algorithm may use additional storage beyond the input list.

  Except where noted above, if any of its preconditions are violated, the `sort` procedure halts the program with a checked run-time error.

- *Architectural design.* My plan is to use merge sort, which guarantees $O(n\ log(n))$ performance.

  Components: Use of merge sort more or less dictates that the sort routine will be composed of two components: a sort function and a merge function. This architecture follows directly from the algorithm:

  - Split an array into halves and sort each half recursively
  - Merge the sorted halves

  There are two base cases: an empty array and an array with one element. These arrays are already sorted and do not need a special component to handle them.

  The solution will use one helper function `merge`, with this specification:

```
void merge(const double *a1, unsigned size1,
           const double *a2, unsigned size2,
           int (*compare)(double x, double y),
           double *result);
  /* write the merger of arrays a1 and a2 into result,
     of which elements 0 through (size1+size2-1) are
     be overwritten */
```

  Parameters `a1` and `size1` specify the first array; `a2` and `size2` specify the second. Both are sorted; neither is mutated. The pointer `result` points to properly allocated (but possibly uninitialized) memory of size at least `sizeof(*result)*(size1+size2)`. If `size1` is nonzero then `a1` is non-NULL, and similarly for `size2` and `a2`.

  The output of `merge` is written indirectly through the `result` pointer. There is no return value; if anything goes wrong, `merge` halts with a checked run-time error.

  Interactions: Procedure `merge` is an independent component. Procedure `sort` calls `merge`, and it also calls itself on arrays that are strictly smaller than the input array (thereby guaranteeing termination).

- *Implementation plan.* I am an experienced C programmer, and I have written merge sort in other languages. A beginning C programmer might take 4 to 8 times as long to carry out the same plan.

  1. A function to print the contents of an array of doubles (5 minutes).
  2. A test that defines these two arrays in memory:

     ```
     double sorted[]   = { 1.0, 2.0, 3.0, 4.0 };
     double unsorted[] = { 4.0, 3.0, 2.0, 1.0 };
     ```

     And a `main` function that prints both arrays. At this point, there will be a trivial end-to-end system that prints. (5 minutes)
  3. Next, write the merge function, and test it by using two statically initialized, sorted arrays (like the ones above) and one *uninitialized* array large enough to hold them both. The `main` test function should include an assertion like

     ```
       assert(NELEMS(a1)+NELEMS(a2) <= NELEMS(result));
     ```

     (15 minutes)
  4. Next, write the sort function, and test it by sorting `unsorted` and reusing my code for printing arrays. (5 minutes)
  5. Write a function to compare two arrays to see if they are structurally equal, that is, they are arrays of the same size, and they contain equal elements in equal positions. (5 minutes)
  6. Change the test so it compares arrays, then prints a message indicating whether after sorting, the two arrays are structurally equal. (2 minutes)
  7. Make the single test into a a test *framework*: a main loop that reads in a list of numbers from the user as well as a list sorted in ascending order. The unsorted numbers are stored in an array, which is passed to the sort procedure. The main loop then compares the result of the sort with the sorted input list for structural equality, then prints the result of the comparison. (15 minutes)

  Testing the merge function first builds confidence in implementing the recursive merge sort. The entire project should take about an hour and should be done on a single day, say, August 14, 2011. About 40% of the time budget is devoted to testing.

- *Test plan.* In a test-as-you-build sort of way, significant unit testing is built into the implementation plan. For overall testing, I will test these kinds of inputs:

  – A list of length 1

  – A two-element list that is initially sorted (e.g., `1 3`) and a two-element list that is initially unsorted (e.g., `3.14 3`).

  - Sorted and unsorted lists whose size is not a power of two, so I can be sure that "split the array into halves" works even when the "halves" are of unequal size.
  - An empty list (array of length 0).
  - A list with negative numbers.
  - A list with multiple copies of the same element (e.g., `1, 1, -3, 3, 3, 2, 2`).
  - A list of 10,000 random numbers, produced by

    ```
    dd if=/dev/urandom bs=10000 count=1 | od -v -t d1 | sed 's/^[0-9]*//'
    ```
  - Lists of 100,000 and 1,000,000 random numbers, so I can use `/usr/bin/time` to be sure behavior is really $O(n\ log(n))$.

Commentary on the design: **Goldilocks and the missing memory**

There are two ways in which the design document above is *not* a model to follow. One simply a mistake commonly made by C programmers. The other, which is more imporant, I'll call the "Goldilocks effect;" it's a problem common to almost all programming methodologies taught in universities.

- *The common mistake.* Almost always, the most difficult part of getting a C program correct is *managing memory allocated on the heap* (with `malloc` and `free`). On this important topic, the document above is completely silent. Don't let this happen to you!

- *The Goldilocks effect.* The design document above is the software equivalent of using an elephant gun to shoot a fly. The mergesort problem doesn't *need* a three-page design document. In fact, unless you are a beginning programmer, you can implement a perfectly lovely mergesort by writing down only the specification of `merge` plus a few test cases. If you are not a beginning programmer, I hope you found the description excruciatingly painful.

  Why do I call this the "Goldilocks effect?" Because this design document is like Goldilocks in the baby bear's chair: it's so big and heavy that it crushes the problem completely, destroying any fun we might have had writing the solution. This sort of thing happens all too frequently, especially in the teaching of that part of software engineering called *programming methodology.*[4] In your Tufts classes, I hope you'll write design documents that are *just right* for the problems you encounter—just as Goldilocks really should have stayed in the mama bear's chair. But the real reason to do a design document is for the big problems—when you have to sit in papa bear's chair. Please remember, then, the purpose of the design document:

[4] I once heard a student complain about a software-engineering course, not at Tufts, by saying "we used all these heavyweight methods for a problem I could have finished in a weekend."

  - To discover early when you are confused
  - To keep you from spending late nights at the computer

  If you find yourself on a programming binge or a debugging binge, step slowly away from the computer and return to your design.

Acknowledgment

This document is adapted from a document written by terrific teaching assistants who worked for Radhika Nagpal and Margo Seltzer; I'm sorry I cannot acknowledge them by name. Thanks to Radhika for permission to adapt it. Any errors or infelicities are my own.