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).