Why We Should Build Software Like We Build Houses

Leslie Lamport
Microsoft research
24 January 2013

Architects draw detailed plans before a brick is laid or a nail is hammered. Programmers and software engineers don't. Can this be why houses seldom collapse and programs often crash?

Blueprints help architects ensure that what they are planning to build will work. Working means more than not collapsing; it means serving the required purpose. Architects and their clients use blueprints to understand what they are going to build before they start building it.

But few programmers write even a rough sketch of what their programs will do before they start coding.

Most programmers regard anything that doesn't generate code to be a waste of time. Thinking doesn't generate code, and writing code without thinking is a recipe for bad code. Before we start to write any piece of code, we should understand what that code is supposed to do. Understanding requires thinking, and thinking is hard. In the words of the cartoonist Dick Guindon:

> Writing is nature's way of letting you know how sloppy your thinking is.

Blueprints help us think clearly about what we're building. Before writing a piece of code, we should write a blueprint.
A blueprint for software is called a specification ("spec").

Many reasons have been given why specifying software is a waste of time. For example: Specs are useless because we can't generate code from them. This is like saying architects should stop drawing blueprints because they still need contractors to do the construction. Other arguments against writing specs can also be answered by applying them to blueprints.

Some programmers argue that the analogy between specs and blueprints is flawed because programs aren't like buildings. They think tearing down walls is hard but changing code is easy, so blueprints of programs aren't necessary.

Wrong! Changing code is hard--especially if we don't want to introduce

bugs.

I recently modified some code I hadn't written to add one tiny feature to a program. Doing that required understanding an interface. It took me over a day with a debugger to find out what I needed to know about the interface--something that would have taken five minutes with a spec. To avoid introducing bugs, I had to understand the consequences of every change I made. The absence of specs made that extremely difficult. Not wanting to find and read thousands of lines of code that might be affected, I spent days figuring out how to change as little of the existing code as possible. In the end, it took me over a week to add or modify 180 lines of code. And this was for a very minor change to the program.

Modifying that program was a small part of a larger task, most of which involved changing code that I had written over 10 years ago. Even though I had little memory of what my code did, modifying it was much easier. The specs I had written made it easy to figure out what I needed to change. Although the changes to my code were an order of magnitude more extensive than the ones to the other code, they took me only twice as long to make.

What do I mean by a specification? It is often thought to be something written in a formal specification language. But formal specification is just one end of a spectrum. We wouldn't draw the kind of blueprints needed for a skyscraper when building a toolshed, nor should we write formal specs for most software. However, it's as silly to write small programs without writing specs as it would be to build a toolshed without first drawing some kind of plan.

These days, the few programs I write are more like bungalows than skyscrapers. I usually specify each method, and most methods are so simple that they can be specified in a sentence or two. Sometimes figuring out exactly what a method should do requires thought, and its spec may be a paragraph or even a couple of pages. I use a simple rule: The spec should say everything one needs to know in order to use the method. After the code has been written and debugged, no one should ever have to read it.

Once I've figured out what a piece of code is supposed to do, the coding is usually straightforward. Sometimes it's not, and it requires a non-trivial algorithm. Getting an algorithm right takes thought, which means writing a spec.

While the specs I write are almost all informal, occasionally a piece of code is sufficiently subtle, or sufficiently critical, that it

should be specified formally--either for precision or for using tools to check it. I've only had to do that about a half dozen times during the past dozen years.

To designers of complex systems, the need for formal specs should be as obvious as the need for blueprints of a skyscraper. But few engineers write specs because they have little time to learn how on the job, and they are unlikely to have learned in school. Some graduate schools teach courses on specification languages, but few teach how to use specification in practice. It's hard to draw blueprints for a skyscraper without ever having drawn one for a toolshed.

Writing is hard, and learning to write takes practice. No simple rules can ensure that you write good specs. One thing to avoid is using code. Code is a bad medium for helping to understand code. Architects don't make their blueprints out of bricks.

The key to understanding complexity is abstraction, which means rising above the code level. The best language for being simple and precise is math--the kind taught in elementary math courses: sets, functions, and simple logic. To make it easier to build tools for checking specs, most formal specification languages add things that are not found in elementary math classes--for example, types. However, the further a language departs from simple math, the more it hinders the abstraction needed to help us understand a complex program or system.

Whether they are formal specifications of complex systems or informal specs of simple code, writing specs improves our programming. It helps us understand what we are doing, which helps eliminate errors. By itself, writing specs won't ensure that our programs never crash. We still need to use methods and tools that have been developed for eliminating coding bugs.

Thinking doesn't guarantee that we won't make mistakes. But not thinking guarantees that we will.