

Why Software Really Fails And What to Do About It

By Chuck Connell

What is software? What is it about software that takes so long to create? And why does software development so often go wrong, compared to other kinds of engineering?

Along with my day job as a software consultant, I like to think about the essence of this stuff I work with every day. I wonder why it is different from other “stuff” that humans build things with, such as bricks and steel and chemicals.

Computer scientists have proposed many ways of looking at software, including as a function with inputs and an output, and as instructions for transforming computer memory from one state to another. I like to think of software, however, as a machine. It is a machine we cannot touch, but it is a machine nonetheless. (I am not the first to suggest this view.) Equivalently, we can think of software as one part of a machine that includes the computer hardware, allowing us to see software as a component of a physical machine.

When we create a new software system we are creating a new machine. The principles that apply to good machine design also apply to good software design, such as durability, maintainability, and simplicity. But, if this is true, why does software development seem harder than mechanical or structural engineering? After all, people build airplanes, bridges, skyscrapers and factories, on time and on budget. (Not always of course, but often.) It is rare that large physical engineering projects are simply abandoned half-built, after years of effort, with millions of dollars spent, left to rust under the elements. But there are many examples of this scale of failure for software projects, including [this](#), [this](#), and [this](#). Many, many others were never made public because their failures were hidden by organizations not wishing to be embarrassed by the scope of their incompetence.

So what, specifically, is it about software that causes large software projects to go wrong more often than with other kinds of engineering? This question has been examined before, including [here](#), [here](#), and [here](#). Some of the proposed answers are poor requirements documentation, and that software is just plain harder than other kinds of engineering. I believe there is another answer, however, which has not been stated clearly.

The problem is not the nature of software; software is just a type of machine and is amenable to known methods for good machine design and project management. The problem is the way we approach software projects and our expectations for their outcomes. In short, we expect too much. Too often we try to “invent the world” with a new software project, instead of relying on well-known designs and methods that are likely to succeed.

Consider this fictional mechanical engineering project that is run like many software projects.

The motivation for this project is that cars are a very poor form of transportation for individual people. This has been widely recognized for a long time. We want a smaller, lighter, cleaner, less expensive device for personal, local transportation.

We will call the new invention Personal Transportation Device 1000, stating its intended selling price in US Dollars. For individual commuting and errands within 50 miles, we want PTD-1000 to make the current automobile obsolete. We do not want the device to use existing, already congested, roads so PTD-1000 will fly.

Our goals include low fuel cost and no pollution, so the motor will be powered by helium fusion. Fusion is an emerging standard, but we believe that this project will provide synergy to fusion research, both driving the research and serving as a test bed for it.

We want the device to be light, which will contribute to efficiency and allow the single user to pick it up, so we will construct PTD-1000 primarily from [Rearden Metal](#). The design team recognizes that the formulation for this metal is not yet finished, so we will assign our A-team of engineers to finish this work in parallel with the other subsystems.

The final key design criterion is that a person who is physically disabled must be able to enter a building after getting there. So PTD-1000 will convert to a wheelchair for use indoors.

The participants in this project all understand that it is a substantial undertaking, but enthusiasm is high for the benefits that will be realized at its completion. There is buy-in by all stakeholders. The investors and engineers have committed to a budget and a completion date of Q4 2011. Everyone has agreed to forgo their vacations for the next year in order to meet this schedule.

Of course, this fictional invention is absurd. Anyone with common sense can see that it will fail dramatically. But take the above scenario, and substitute software concepts for the physical details, and you have an accurate description of many real software projects.

In contrast to software projects, traditional engineering relies on tried-and-true materials and methods, applied in well-understood ways to new applications. An aeronautical engineer may

design a new aircraft that does not look or function exactly like any previous plane, but the structural members, outer skin, wiring components, and engines usually are copies of proven parts and techniques. Manufacturing plants, skyscrapers, roads, and bridges are built with a similar philosophy – substantial reuse of known pieces, put together largely in known ways.

Traditional engineers, of course, sometimes advance the state of their art, by using new materials or techniques or both. The Burj Khalifa recently set a new height record for buildings. The engineers did this partly by employing a [special formulation of concrete](#) to achieve the weight-bearing strength required, and by pouring the concrete at night to aid in proper hardening. But these innovations are in tightly constrained situations, making incremental improvements to well-studied topics. Concrete has been around for [thousands of years](#) in some format and 200+ years in its modern version. New bridges similarly use known methods, with possibly some incremental innovations to solve special problems for that location. The public would not have it any other way.

Software projects often take the opposite approach, attempting to use many radical new materials (data structures) operated on by grossly unproven methods (programming code) to produce a machine that performs a never-before-seen function. Our large software projects often resemble precisely the sort of absurd dream invention described above. Why do we do this? The reason is that since we cannot see or touch software, it appears that all the parts of software, tested or novel, are quite similar and can be used equally well as machine subassemblies. In fact, software components do not all have the same approximate reliability, not by a long shot. We fail to appreciate fully that well-tested software modules can have a high probability of successful reuse, while novel software components are often a crap-shoot.

When we visualize software as a machine, it becomes clear just how unwise it is to invent too much in a new software system. Picture the overall software as a factory assembly line of robots, or a new kind of automobile. The major software modules are sections of the factory, or important pieces of the automobile. The software subroutines are parts making up the larger mechanical components. Individual lines of source code are single pieces of metal in a robot, or springs, or gears, or levers. Function parameters are rods or lasers reaching into another mechanical subassembly. When the assembly line or car is started for the first time, the parts may not work together correctly. They may rub or bang into each other, preventing the whole machine from working right. This might occur in hundreds of places. Some problems may not be seen until a certain sequence of actions is attempted at the same time.

In the same way, a large software project is an incredibly complex machine, with millions of possible interactions among overlapping parts, compounded by interactions of the interactions. The full behavior of many software systems is well beyond human understanding. This is why

we cannot accurately predict bugs in complex software; we are trying to build machines we cannot comprehend.

Oversimplifying a bit, there are two common approaches to software projects.

1. Design and build software in a conservative manner, using tried-and-true components, assembled by a stable team of engineers, who have successfully built similar systems. These projects usually can be estimated accurately, and completed on time and budget.
2. Attempt to create software that is substantially new. These are really research projects, not engineering endeavors. They have uncertain outcomes and no reliable time/cost estimates.

An example of #1 is the creation of a new compiler by a software development company that has produced many compilers for dozens of languages and target machines. When this company takes on a new compiler project, for a variation of an existing source language, with a carefully specified target instruction set, by an experienced team of compiler engineers, then this project has a high likelihood of success. Techniques such as reusable class libraries and design patterns help software projects conform to this model.

An example of #2 was the FBI's Virtual Case File, [cited above](#). No one had ever created a software system to perform the functions envisioned for it. Creating it was like trying to construct a wholly new type of machine, from a new kind of metal, using a yet-to-be-invented welding technique.

Either of these two approaches to software is valid. The key problem is that we take on projects like #2, but pretend they are like #1. This is what ails the world of software development.

We fool ourselves about how well we understand the complex new software machines we are trying to build. Just because we plan to code a new project in a known programming language, say Java, and our engineers are good at Java, this does not mean we have answers to all the challenges that will arise in the project. Using the mechanical analogy, just because our inventors have put together many machines that use springs and gear and levers, does not mean we can correctly build any machine using these parts.

We can't have it both ways. If we want an accurate budget and completion time, we cannot engage in significant research during a software project. Conversely, there is nothing wrong with research and trial-and-error, but we should not think we know when it will be finished.

But what is the solution in the real world? Everyone would like to make software engineering as predictable as traditional engineering. There are many important pending software projects with large unknowns. We cannot simply say, "Oh, this software poses some new challenges, so

let's give up." The solution is to get over our hubris that software development is some special kind of animal, unlike other engineering endeavors, and that we programmers are so much smarter than our traditional engineering brethren. Software is just a machine, and people have been building machines for a very long time.

To wit, here is my prescription for improving the success rate and reputation of software developers...

- Stop fooling ourselves about how much we know and how clever we are. Large software projects are impossible to understand fully. No one can grasp all of the overlapping effects of each component. Picturing software as a physical machine helps to illuminate just how complex these systems are.

In a large software project, there may be one person who fully understands each particular component, but that is a different person for each component. No one has a grasp of the whole system, and we have no way to meld isolated individual knowledge into a collective whole. This was precisely the problem with the embarrassing tale of the [Metric-English measurement error](#) on the Mars Climate Orbiter. (Wasting \$300 million in taxpayer money.)

- Incremental improvement to existing systems is good. Sometimes this means adding one additional feature to a working system. Sometimes it means combining two working systems with a new interface. And it is helpful if the interface method itself has been used elsewhere.
- Iterative development is good. This applies the above principle, again and again, to one particular software system. The first release of the software does little more than say "Hello" to the user. The next release adds one basic feature. The next, one more, etc. The idea is that each software release only has one major problem to solve. If it does not work, there is one thing to fix. Each release is an incremental improvement to working software. In practice, of course, we may stretch a bit and include a few new features in each release, but we never attempt to create a huge, complex piece of software all at once. (See the [Agile Manifesto](#).) The iterative approach also can be applied to [estimating the size](#) of software projects.
- Research projects are great, but be honest about them. The Denver Airport software disaster, [cited above](#), could have been avoided if it were handled in this way...

Admit that we don't know how to sort airline baggage automatically, but would like to

solve this problem. Start such a research project in an empty warehouse, using a few conveyor belts, some bar-coded suitcases, and some sorting gates with embedded software. After working out the kinks, try the system at a small airport, for incoming flights from one other city. When that works, try it for all flights to this small airport. After success there, and further hardware/software refinements, create a similar system at a mid-size airport. Improve the hardware/software again. Install at several mid-size airports and fix any problems.

Then you are finally in a position to say, “Let’s think about handling baggage at a large airport this way.”

Software development need not be a mystical process, undertaken only by the most brilliant, with no hope of predicting the outcome. Software is a machine, and over many years we have learned the principles of good machine design. Unfortunately, because software is so new and is impossible to see or touch, we get clouds in our eyes when we think about software projects. We forget that we know how to plan, design, and construct high-quality machines – by incremental improvement to previous machines, using proven materials and methods.

Chuck Connell is a software consultant and writer in Bedford, MA. He can be reached at www.BeautifulSoftware.com.