Why Bad Things Happen to Big Software Projects

(And How to Prevent Them from Happening to Yours)

Chuck Connell, BeautifulSoftware.com

There are many troubled software projects in the world right now. There always are. If you ask the programmers and line managers for one of these projects why things are going wrong, they will probably say something like…

- *We don't have enough time and money.*
- *Upper management is trying to squeeze a year of work into every quarter, and they are making each of us do the work of two people.*
- *If we just had a more realistic schedule and more help, we could get this thing done.*

It certainly is true that some software projects fail because of too little time and money. Startup companies often face the challenge of finishing a commercial-grade program with limited resources.

But lack of time and money is not the reason for the colossal software failures that make headlines. The reason for truly expensive failures is actually the opposite. Big software projects usually go off track because they have too large a budget, too many people working on them and are too important.

Examples of this kind of error are well known.

- The U.S. Federal Aviation Administration's re-engineering of its air traffic control system, a project known as Advanced Automation System (AAS), cancelled after $6 billion spent.
- The Denver airport's automated baggage handling system, cancelled after $500 million in outlays and delay cost.

- The infamous U.S. FBI Virtual Case File system, designed to “connect the dots” among disparate pieces of intelligence, cancelled after $170 million spent.

- The U.K. National Health Service’s automation of its patient medical records, via the Lorenzo software, is in deep trouble as of this writing and will likely be scrapped after $24 billion spent.

The failure of these projects flowed directly from the following assumptions (explicit or implicit) at their outset.

1. *This project is really important. Management will never cancel it.*

2. *This project will have such a huge benefit when finished that it is OK to blow past our original budget and ask for more time, money and people. We will get them. If we skate past the new targets, we can ask for more again, and we will get it. And again.*

3. *Key aspects of what we are trying to do are so novel that we have to invent new methods for doing them. We are not just writing a big program; we are breaking new ground.*

4. *The software we are going to write will be very complicated. But that’s OK; the problem itself is complicated.*

These projects were unsuccessful because they tried to do too much, tried to invent too much from scratch, wrote too many lines of code and had a vision that was too grand. While humans are smart in some ways, no person can grasp all the interactions in such highly complex systems, leading to almost inevitable failure. These broken software projects were not built carefully out of known, working components. Instead, their designers attempted to spring them into existence from whole cloth.

Combining a sense of entitlement, a lack of limits, a never-before-solved problem and acceptance of complexity is a recipe for disaster. These projects were designed for failure, and then participants were shocked to discover how badly (and expensively) software can go wrong. It was actually fairly easy to predict on Day 1 that the projects would go belly up.

So what is the solution? Never do a big project? Almost. The answer is not to do them in the same way. The right approach for very large software projects is to adopt a change in mindset and a change in architecture.

New Mindset

Every project has a budget that is not worth exceeding. No project is so important it is worth any cost.

To take one example, streamlining U.S. Internal Revenue Service operations would certainly have a payoff. The IRS now spends about $12 billion per year to collect about $2.3 trillion in taxes. If a better computer system allowed for just an additional 2% in collections, say through reduced fraud, this would amount to an extra $46 billion income per year. So it would surely be worth a billion dollars to buy/build such a new computer system for the IRS. It would even be worth $40 billion dollars, since the payback period would be just one year. But would it be worth $500 billion dollars? No, since the payback period would be more than 10 years and by then the system requirements and available technology likely would change.

No project participant, at any level, should ever think that he/she is immune from limits or failure. Coming to work every day in a state of high fear is not good for productivity. But coming to work each day with the belief that efficiency and productivity do not matter is inviting a bloated project, as all of the above examples were.

The proof that every project has time/budget/resource limits is the fact that the first three projects cited above were ultimately cancelled, despite their importance, with the UK health records effort close behind. (IRS modernization keeps floundering also, but has been restarted several times with different players.)

New Architecture

A complex monolithic project, which must all come together in order to work, is a blueprint for problems.

Instead, break apart large projects so they are in smaller, more manageable pieces. The overall architecture should specify separable components that can succeed on their own or in combinations with their peers, performing useful operations before the whole project is complete.

Note that I am not repeating the tried-and-true advice to break a program into modules or classes; any good design does that. The new architecture advocated here divides a large software system into separate components such that *each, on their own, performs useful work.* When the components are combined with other working components, even more useful work is performed.

The overall vision is a set of working parts, which *do not all need to be completed* in order for the system to be useful. The system is survivable without all of its pieces.

Of course, in these systems, the modest-size pieces may still be considerable software projects, but their size and complexity is limited to a realm where they have a reasonably high probability of operating correctly in a predictable amount of time. Since the overall system is designed to be useful with various combinations of components, the lack of one part is not a show stopper. There is no single point of failure or monolithic “all or nothing” nature to the solution.

**

Many of the world’s giant software projects have been disasters waiting to happen, right from the start. Instead, participants on large software projects should adopt two new credos:

1. *We need to work smart or we could lose this project.*

2. *Let’s make modest*-size *useful pieces that can be combined in useful ways.*

This is a recipe for success.

Chuck Connell knows software projects from all sides. He has been a hands-on programmer, a software product architect, a manager of programmers, a university teacher of software engineering and a consultant. His consulting practice BeautifulSoftware.com helps organizations assess the health of software projects, manage ongoing projects and turn around troubled projects.

For more information….

http://spectrum.ieee.org/computing/software/why-software-fails (article about software failures)

http://calleam.com/WTPF/?page_id=3 (ongoing list of software failures)

http://www.oig.dot.gov/sites/dot/files/pdfdocs/av1998113.pdf (report about FAA AAS)