Software Runaways — Some Surprising Findings

Robert L. Glass
President
Computing Trends

I have always been interested in computing projects that failed. There seems to be a much more indelible lesson to be learned from failures than from successes.

Some of you readers may know that I once wrote a column for a leading computing newspaper under an assumed name, and the content of the column was fictionalized failure stories. (I wrote it under an assumed name and fictionalized the stories because I suspected that my employer at the time would not appreciate my washing its dirty linen in public!) Those columns eventually became a book, *The Universal Elixir and Other Computing Projects That Failed*, and then grew to fill another book, *The Second Coming: More Computing Projects That Failed*.

Buoyed by that success, I did it again in *Computing Catastrophes*, true stories of mainframe computing projects that failed and yet again in *Computing Shakeout*, true stories of microcomputer projects that failed. I was on a roll. Failure was my fate!

But the last of those books was (self-) published in 1987, over a decade ago. It was not that I lost interest in failure; it was simply getting harder to find good failure stories! The frequency of early-day failures, and the shakeouts in the mainframe and microcomputer industries, had largely ended. Computing had become an almost boringly successful field.

Some would, of course, take issue with that statement. Those who cry "software crisis" promote the belief that the software field is full of failure, with far too few successes in between. But that is not how I see it. In spite of the time I've spent finding and publishing failure stories, I believe that software is the dramatic success story of our age, the spark that ignites the computing era. There have been failures, of course; but what makes them interesting is partly that there are not that many of them. It is what journalists and software people call "exception reporting" — we tend to focus on the things that go wrong because they are

more interesting or important than the run-of-the-mill things that go right.

Well, I am at it again! I have been gathering more stories about software runaways for about ten years now, and I have enough of them to put together yet another book. The book will contain stories about the Denver Airport Baggage Handling System, the FAA Air Traffic Control System, the Internal Revenue Service Modernization effort, and a baker's dozen or so other examples of software projects that got way out of control, most of them crashing and burning (usually figuratively rather than literally!) Once a failure nut, always a failure nut!

One of the things I do as I accumulate stories for a book is to analyze the lessons they teach us. It is a good way to create an outline for the book, for one thing; and it is also a way to make sure that there's value added for the reader. These are not only fun stories to read, but the lessons make the reading a learning experience.

What I would like to do in this column is share with you some of the learning experiences from that forthcoming book which will be called *Software Runaways*. However, at this writing the identity of the publisher cannot be disclosed. I think there are some particularly interesting things happening in our field, as reflected in these failures; and they do not always agree with what our textbooks and our research papers are telling us.

First, let's get the predictable out of the way. Here are some things I discovered that match what textbooks and research are telling us:

1. Most of the runaway projects are (or were) huge. It is well known in the field that huge projects are much more troublesome than smaller ones. These stories support that finding.
2. Most runaway projects result from a multiplicity of causes. There may or may not be one dominant cause, but there are always several problems contributing to the runaway.
3. Many of the runaway projects were lauded early in their history as being "breakthroughs," significant advances over the systems they were replacing. It appeared that visibility into the possibility of failure did not emerge until the project was well under way.

But the unpredictable is much more fascinating. Here are some characteristics of these runaway projects that none of my reading (and I suspect yours as well) prepared me for:

1. Technology was just as often a cause of failure as management. The literature, especially in the software engineering field, tends to say that major failures are usually due to management. But for nearly half of these 16 failure stories the dominant problem was technical. (We will return to this thought later).
2. There were two especially surprising and dominant technical problems. The first was the use of new technology. Four of the projects, fully expecting to be breakthroughs because they were using the latest in software engineering concepts, instead failed because of them!

   One used a megapackage approach to replacing its old, legacy software, betting the company on the approach, and losing. Another used a fourth generation language for a large project, and found (after the work was complete!) that it was not capable of meeting performance goals for the (on-line) system. Yet another tried to port an existing mainframe system to client/server, and found the complexity increase got out of hand. And the fourth put together so many new technologies that the project foundered from their sheer weight (that did not prevent some of the project's principals from proclaiming the project a success in their company's house organ!)
3. The second dominant technical problem was performance. Many of these runaway projects were in some sense real-time (that in itself is a pertinent finding), and all too often the as-built systems simply were too slow to be useful. This is particularly interesting because most

at least, suggest that the field may have overplayed that hand.

There is one more finding that I am not sure what to make of. A fairly large number of these runaway projects had something else in common. They were about a collection of applications that might be characterized as the domain "movement of goods." The Denver Airport Baggage Handling System is an obvious case in point. Two others were warehousing systems. Perhaps there is something in this class of applications that should sound a warning signal. It is an interesting speculation, one to be watched closely by those about to embark on such a project.

While putting this latest book together, I came across a relevant research paper (Cole, 1995) published outside the mainstream Information Systems and Software Engineering literature. It was a recent KPMG study of runaway projects (using a survey of IS managers), and it did a nice job of characterizing those projects.

Here are some of the trends identified in that paper:

1. Respondents believe that runaways are decreasing in number (42% said they believed the frequency of runaways was decreasing, whereas only 8% saw them increasing)
2. Packaged software seemed equally as likely as custom software to be involved in a runaway (47% used a mixture, 22% involved packages alone, and 24% were custom).
3. Schedule was most frequently the prime runaway problem (89%); 62% had cost overruns.
4. Most of the systems were misconceived from the beginning (more that half began to show runaway symptoms during the systems development phase). Note that this conflicts somewhat with my own findings discussed above.
5. Most runaways are not discovered by senior management (19%), but rather by the project team itself (72%).
6. Technology is rapidly increasing as a cause of runaways. The paper cited an earlier (1989) KPMG study that found technology a problem only 7% of the time; but in this more recent study 45% had problems with new technology, and 16% felt they had used the wrong technology. The author of the KMPG paper concludes, "Technology is developing faster than the skills of the developers." I'm not so sure. It seems equally possible that "Technology advances do not always work as claimed."
7. The leading causes of runaways are (in order of decreasing importance):
   a. Project objectives not fully specified.
   b. Bad planning and estimating.
   c. Technology new to the organization.
   d. Inadequate or no management methodology
   e. Insufficient senior staff on the team.
   f. Poor performance by hardware and/or software suppliers.

The runaways in my book map fairly nicely onto this collection of causes; I have one to four examples for each of the above categories. The only major difference is that performance problems were a significant cause in my stories, and that category was not included in the paper's research findings.

In summary, I would like to say this: software runaways don't occur often in our field, but when they do they are increasingly more visible. Many of the stories in my book were mentioned one or several times on TV and in general print media news reports. And a surprising number of these projects, as evidenced both by the research findings I cited, and by the stories I have collected, were flawed because of the technical (not just the management) approach. There are some words of warning here for those who embark on major software projects; none of us would like to see our best efforts become the headline story on the nightly news!

Reference

Cole, Andy (1995). "Runaway Projects: Cause and Effects," *Software World* (UK) Vol. 26 No. 3.

About the Author

Robert L. Glass is president of Computing Trends, publisher of *The Software Practitioner,* and *PERC (Practical Emerging Research Concepts) in Information Technology.* He also writes the "Practical Programmer" column for the *Communications of the ACM.* He has been active in the field of computing and software for over 40 years, largely in industry (aerospace) but also as an academic (Seattle University, Software Engineering Institute). The author of some 20 books on computing subjects (many of them about failure!), he describes himself as having his head in the academic side of computing but his heart in its practice.