

Major Causes of Software Project Failures

Lorin J. May

CROSSTALK Associate Editor

Most software projects can be considered at least partial failures because few projects meet all their cost, schedule, quality, or requirements objectives. Failures are rarely caused by mysterious causes, but these causes are usually discovered post-mortem, or only after it is too late to change direction. This article is based on interviews with software consultants and practitioners who were asked to provide "autopsies" of failed projects with which they have been acquainted. Although not a comprehensive compilation of failure causes, this article outlines several areas that should demand your attention.

A few years ago marked the rollout of what could have been called a Titanic of military projects, except the original Titanic was ahead of schedule when it sank. Hundreds of millions of dollars over budget and years behind schedule, the first phase of this huge military system was finally "tossed over the wall" and over the top of a network of separate programs used by thousands of practitioners. Although long hampered by quality problems, big hopes were again riding on the system once it passed acceptance testing.

The intended users refused to use the system. It lacked features they said were essential to their jobs while requiring steps they considered unnecessary or burdensome. The project eventually died a visible, painful death amid litigation and congressional inquiries.

This failed project was not atypical of chronic problems in the software industry. According to the Standish Group [1], in 1995, U.S. government and businesses spent approximately \$81 billion on canceled software projects, and another \$59 billion for budget overruns. Their survey claimed that in the United States, only about one-sixth of all projects were completed on time and within budget, nearly one third of all projects were canceled outright, and well over half were considered "challenged." Of the challenged or canceled projects, the average project was 189 percent over budget, 222 percent behind schedule, and contained only 61 percent of the originally specified features.

Other studies have likewise concluded that failure is rampant, although not necessarily to the same degree. One

reason for the varied conclusions is that most failed projects are never studied—even by the organization that experienced the failure. Having wasted so much on a fruitless venture, few organizations will invest more time or money to collect and analyze additional data, whereas any data that had been collected may be massaged or hidden to protect careers or reputations. Thus, information about project failures often relies heavily on subjective assessments. This article is no exception.

For this article, a failure is defined as any software project with severe cost or schedule overruns, quality problems, or that suffers outright cancellation. It is based on interviews with practitioners and consultants who were asked to describe the causes of software project failures with which they have been acquainted. If there is anything notable about the interviewees' diagnoses, perhaps it is that many of these problems have been documented for years, but somehow they keep cropping up. Also worth noting is that most of the failure causes mentioned originate before the first line of code has been written. The failure causes are listed in no particular order.

Poor User Input

Although the Titanic project mentioned earlier was riddled with problems, it ultimately failed because the system did not meet user needs. According to Paul Hewitt, a consultant with the Software Technology Support Center (STSC), the acquirers and developers of this system had received most of their requirements from higher-level supervisors and so-called "users" who were not regularly

using the existing system. Although "not invented here" syndrome contributed to the system's eventual lack of acceptance, the bottom line is that the system was inadequate for its environment.

By contrast, Hewitt has observed successful programs in which "end users and developers [were] working together in the same cubicle." Although this is not always possible, Hewitt said projects are likely headed for trouble unless informed end users are giving meaningful input during every phase of requirements elicitation, product design, and building. The input needed by these users has less to do with issues like screen layouts than with how the system would be used in the field, according to Michael Allen Latta, chief executive officer of Ohana Technologies Corp. in Lafayette, Colo. He said the user should be asking, "How do I use it over time? Does it provide the right tools? What do I put into it, and what do I get out?"

However, there can also be problems if the users are too close to the requirements. Shari Lawrence Pfleeger, president of Systems/Software in Washington, D.C., had just started consulting on a large federal system acquisition when she started to study its requirements, which were supposedly "clean" due to the input of highly knowledgeable users. Even without any prior understanding of the system or its field environment, Pfleeger needed only a few hours to see that the requirements were full of hidden assumptions and conflicts.

"[The users] didn't think of the consequences of what they were requiring," she said. "They assumed that how things were done in the past was how they would always be done in the future."

The users assumed the elicitors understood more than they did about the users' jobs, but this was not entirely the users' fault. *All* involved parties, including the developers, must understand the business of the other parties. This need continues throughout development process. Without this understanding, the parties "don't even know what questions to ask," Pfleeger said, and important issues fall between the cracks.

Stakeholder Conflicts

A few years ago, a major airline, rental car company, and some hotel chains created an incentive plan to give customers frequent flier-type points to "cash in" for any of the participating companies' services. They commissioned a complex software system to track points and compensation. Sometime later, the software developers needed some clarifications, i.e., with input A, does the system choose X, Y, or Z? The stakeholders could not agree on the answers. Forced to acknowledge deep incompatibilities among their business interests, the system was canceled in an expensive, litigious failure of the entire enterprise.

The stakeholders had worked under "the illusion that everyone was going to get everything that they wanted," explained Tom DeMarco, principle of the Atlantic Systems Guild. They "papered over their differences" rather than going through conflict resolution in the early stages. Their differences were exposed by the developers because "coders cannot make an ambiguous system."

Stakeholder conflicts can play many different roles project failures. For example, "some projects are ultimately canceled because people don't like each other," said Capers Jones, chairman of Software Productivity Research, Inc.

Other projects fail because the developers do not know who the "real" stakeholders are, according to Ed Yourdon, chairman of the Cutter Consortium. Yourdon worked with a large mutual fund company that had been working on a \$300 million software system. The developers had been working closely with the information technology vice president, who was perceived to be the primary stakeholder for

the system. When the system ran into some problems, it drew the attention of the chief executive officer, who turned out to be the real stakeholder in the system even though he had not previously been involved with it. After seeing the involved risks, he immediately withdrew his support for the system.

"No one bothered to ensure that he was going to support it," Yourdon explained. "No one made him aware of problems while it was being developed." Yourdon says many projects fail because the project leaders do not have a sense of who will ultimately declare whether a project is a success or failure, and then they are "blindsided." He said the true stakeholders need to hear good and bad news in "small pieces" rather than in "one chunk."

Other projects, especially smaller projects within larger projects, never go anywhere because the internal stakeholders never agree on priorities. Watts Humphrey, a fellow at the Software Engineering Institute, calls these "pretend projects," meaning a few developers work on them half time or quarter time, and nothing is ever delivered.

"They are kidding themselves that they are working on [these projects]," Humphrey said. "No one can work quarter time on a project. ... They haven't faced the need as a management team to decide what they are really going to do with it. They need to put real resources on it" rather than merely pretend the project is under way.

Vague Requirements

Mariea Datz, president of Peripheral Visions in Houston, Texas, learned a hard lesson about what happens when a project is started while the requirements are nebulous. The U.S. division of an oil company hired Datz's company to create the "first draft" of a program so that they could impress their European counterparts and justify further funding. But the oil company officials only had a general idea of what the program was to do and tried to revise and refine their ideas while Datz's company was working on the program.

"For every step we would take, we'd go three backward," Datz said. "We

would start down one path and then have to stop and go down another." Project cost and quality quickly went out of control, her company was blamed, and she lost the contract to finish the job. Like many failed projects, the scope had not been narrowed enough at the outset to have led to any reasonable chance for success.

One obvious solution is to establish a reasonably stable requirements baseline before any other work goes forward. But even when this is done, requirements will still continue to creep. "You can't design a process that assumes [requirements] are stable," advises Humphrey. In virtually all projects, there will be some degree of "learning what the requirements really are while building the product," he said. Projects could be headed for trouble if architectures and processes are not change-friendly, or if there are poorly established guidelines that determine how and when requirements can be added, removed, and implemented—and who will shoulder the cost of the changes.

Poor Cost and Schedule Estimation

It is unfair to call a project a failure if it fails to meet budget and schedule goals that were inherently unattainable. Like all engineering endeavors, every software project has a minimum achievable schedule and cost. Fredrick Brooks summarized this law in *The Mythical Man Month* [2] when he stated, "The bearing of a child takes nine months, no matter how many women are assigned." Attempts to circumvent a project's natural minimum limits will backfire.

This problem occurs any time someone "makes up a number and won't listen to anyone about how long other projects took," said Jones. According to DeMarco, projects are often intentionally underbid because of the "attitude that putting a development team under sufficient pressure can get them to deliver almost anything."

The opposite is what usually happens. For example, if a program should realistically take five programmers one year to complete, but instead you are given four programmers and eight

months, you will have to skimp on design time and on quality checks to reach project milestones.

“Cutting a corner that undermines the entire foundation of the project is not cutting the corner,” states Robert Gezelter, a software consultant in Flushing, New York. “There will be heavily disproportionate costs downstream.” Skimping leads to weak designs, dramatically higher defect densities, much more rework, and virtually endless testing. In the end, the project will cost more, take longer, and have worse quality than would have been possible if a realistic schedule and budget had been followed.

According to Jones, this problem can be easily remedied. Several estimation tools on the market can combine numerous variables to provide realistic estimates within a few hours [3], even at the early critical decision-making junctures—before requirements are firm.

Skills that Do Not Match the Job

Decades ago, Morris Dovey, information director for Check Control, Inc. in West Des Moines, Iowa, worked on major government software contracts before becoming so frustrated he decided to never work with government contracting again.

“It was being made artificially difficult,” Dovey said. The technologists had to endure what he considered avoidable delays and mistakes because “decisions were being made by people with no technical expertise in the area” but had all the authority.

Latta warns that managers can perform poorly if they lead projects that do not match their strengths. “Projects dealing with high technology need managers with solid technical skills,” Latta advises. In such projects, authority must reside with people who understand the implications of specific technical risks.

However, the best technologists are not necessarily always poised to be the best managers. “The skill set for management and programming are disjoint,” Jones observed. The larger the project, the more need there is for people with excellent planning, oversight, organization, and communications skills; excel-

lent technologists do not necessarily have these abilities.

Skill-driven challenges are not limited to management. Poor developers can sap productivity and make critical, expensive errors. Generalists can also poorly perform duties better left to specialists, such as metrics experts or testers.

The solution to skill-driven challenges is easy to define but difficult and expensive to accomplish: Attract and retain the most highly skilled and productive people. “Knowledge is money,” noted Tom Pennington, senior network manager for The MIL Corporation in Arlington, Va. However, there is an eventual payback. Pennington believes a team made up of higher-paid people with the right specialized skills is worth far more per dollar to an organization than a group of lower-cost people who need weeks or months of fumbling through a new process or technology before they can start being productive.

“You get what you pay for,” Datz echos. “You’ll also pay for what you get.”

Jones advises that “if you can’t get the best ‘techie,’ get the best managers.” He said good managers can often get above-average results from average employees, whereas great employees can have much of their potential squandered by mediocre management.

Hidden Costs of Going “Lean and Mean”

DeMarco believes project managers and technologists are often unfairly blamed for problems caused by people “two levels higher.” He believes managers and technologists are generally competent and getting better every year, but they are “goaded” into overtime work because of “the 1990s stupid flirtation with lean and mean”—cutting jobs and expecting the same work with fewer people and less money, whether such a feat is possible or not. DeMarco says the often-intentional “dishonest pricing” of projects is often off by a factor of two or four or more, requiring never-before-seen levels of performance.

“Any failure will be viewed as a direct result of underperformance,” he charges, even though underperformance is “not even a significant factor” in the failure of

most projects. Instead, he says, the failed projects simply had goals that were inherently unattainable.

Humphrey has observed a different “lean and mean” problem. In many “downsized” organizations, he says, developers are doing their own expense accounts, clerical work, software updates, and other duties—and at a higher labor rate and with less skill than could be performed by support specialists.

He estimates that many software developers are spending half their work hours slowly plodding through tasks that have nothing to do with developing software. “Software people are very unskilled clerks,” he said. “It’s an enormous productivity issue.”

Failure to Plan

Humphrey took charge of commercial software development for IBM at a point when the company was taking too long to finish projects and was missing all its announced deadlines. “People were working hard, but no one had plans ... because no one required them to make plans,” Humphrey recalls. In response, he required that a detailed plan be developed before any release date was announced. For the next two and one half years, the division never missed an announced date.

“If software developers built bridges, we’d show up at the site with some scrap iron and say, ‘let’s start building!’” quipped Reuel Alder, a manager at the STSC. Alder agrees that inadequate planning is a major reason software projects spin out of control.

Humphrey said project managers often do not plan because “any plan they put together won’t meet the [desired release] date, so they can’t plan.” Even though detailed planning saves an enormous amount of time in the long run, Humphrey says many other managers and developers believe it to be unnecessary. “They think time spent on things like planning, design, requirements, and inspection gets in the way of real work, which is coding and testing,” he said. “This comes from the view of programming that the issue is to get the software out the door. But there’s a difference between speed and progress.”

“We need a lot fewer heroes,” adds Gezelter. He believes organization “heroics” would frequently be unnecessary if projects had been properly planned. “We keep rewarding people for charging off on suicide missions,” he said.

Communication Breakdowns

When Pfleeger was asked to consult on a large project that was in trouble, she asked the managers to develop a process model for the project. She did not necessarily want the model for her own use, but wanted the managers to talk to the developers. Once they did, they realized the project had gotten so large that the same code was being tested by two teams that did not know the other existed.

Such problems are common on large projects, especially if people are working at different sites. In many troubled projects, “there isn’t one person who has an overview of the whole project,” she said. Especially on large projects, Pfleeger advises that additional time be taken periodically to have everyone in every position learn the big picture. “The people working on the pieces need to know how their one piece fits into the entire architecture.”

Poor Architecture

Pfleeger says an example of flexible architecture is the Patriot missile used during the Gulf War. It was not designed to intercept scud missiles, but the software was able to be reconfigured to support the new function. On the other end of the flexibility spectrum was a security program created to protect sensitive word-processing documents. Everything worked well for a few months until the operating system was updated. The word-processing programs still worked, but the security program became useless and unfixable because much of its code was tied to operating system features that were dropped in the new system.

“People didn’t think ahead about what was likely to change,” Pfleeger said. Architecture must allow for organization and mission changes.

Gezelter said software developers often build with no more forethought than the man who built a beautiful boat

in his workshop and then could not get it out the door. “If you do [architecture] right, no one will ever realize it,” he said. “But if you do it wrong, you will suffer death by a thousand cuts. Bad choices show up as long-term limitations, aggravation, and costs.”

Gezelter suggests viewing software architecture like house-building: “Plumb” and “wire” for features and additions you have not thought of yet. Then, when unanticipated needs or business changes arise, you can add or modify without performing the software equivalent of “ripping apart the walls and rebuilding them again.”

Late Failure Warning Signals

Does the following scenario by Yourdon seem familiar? A schedule and budget are determined “by edict by people you were afraid to say no to,” and it is politically unwise either to say or show the estimate is far from achievable. All your early milestones involve diagrams, designs, and other documents that do not involve working code. These and other project milestones then go by more or less on schedule—at least as far as upper management can tell—and testing starts more or less on time. Not until the project is a few weeks from deadline does anyone dare inform the “edict makers” that at the current defect detection rate, the project will not be completed even close to its deadline.

“Nobody seems to acknowledge that disaster is approaching,” Yourdon said, even among people who sense there is a problem. “There is no early warning signal.” Until more organizations abandon waterfall-style development in favor of processes that demand early working code or prototypes, he says this scenario will continue to be familiar.

Yourdon says the above problem is also extremely common with year 2000 work. He believes many year 2000 conversion teams, if they were allowed, would say of their current situation: “Within this limited time and pitiful budget and understaffed team, sure, we can deliver it on time—with a million bugs in it.”

In a perfect world, lower-level people could convince upper-level managers

that their edicts are unworkable before the project got under way. But until this happens, Yourdon says development cycles need to be adopted that allow you, at the earliest possible moment, to “provide evidence that [the project] is or is not working.”

Conclusion

Other causes of failure could be added ad nauseam, but the existence of additional factors is not the point. As Jones noted, “There are myriad ways to fail. ... There are only a very few ways to succeed.” [3] The factors of successful project management have been documented for years—they merely need greater attention. But if this article has helped serve as a reality check for your project, it will have served its purpose. If you violate any of the principles noted by the consultants and practitioners in this article, you should not expect to succeed in spite of yourself. ♦

About the Author



Lorin J. May is an editor and columnist for *CROSSTALK: The Journal of Defense Software Engineering*. He is employed by NCI Information Systems,

Inc., under contract to *CROSSTALK* at the Software Technology Support Center. He was previously an editor for two book publishers and was a part-time freelance writer. He has a bachelor’s degree in journalism from Weber State University in Ogden, Utah.

OO-ALC/TISE
7278 Fourth Street
Hill AFB, UT 84056-5205
Voice: 801-777-9239 DSN 777-9239
Fax: 801-777-8069 DSN 777-8069
E-mail: MayL@software.hill.af.mil

References

1. The Standish Group, “Chaos,” 1995, <http://www.standishgroup.com/chaos.html>.
2. Brooks Jr., Frederick P., *The Mythical Man-Month* (20th Anniversary Edition), Addison-Wesley, Reading, Mass., 1995.
3. Jones, Capers, *Patterns of Software Systems Failure and Success*, International Thompson Computer Press, Boston, Mass., 1996.