CAPERS JONES, SOFTWARE PRODUCTIVITY RESEARCH

Our Worst Current Development Practices

How to get people and technology to work together.

There's an old saying that goes: "There are two kinds of failures: those who thought and never did and those who did and never thought." In the software business, our worst failures tend to fall into the second category. Too often we plunge into a major project, without considering the factors that have the strongest bearing on success or failure. Even worse, many software managers refuse to learn from failure, repeating the same destructive behavior project after project, then wondering why Mylanta has become one of their major food groups.

This issue, Capers Jones talks about failed projects and the practices that lead to them. Throughout his career, Jones has studied thousands of software projects. His strength lies in his ability to analyze the critical software metrics and extract pragmatic advice for real managers on real projects. Heeding his message may help you to think first, then do.

— Roger Pressman

Editor:
Roger Pressman
R.S. Pressman & Associates, Inc.
Orange, CT 06477
pressman@rspa.com
http://www.rspa.com

FOR MANY YEARS I HAVE STUDIED TWO extremes associated with software-development: Projects that set new records for quality and productivity and those that were total disasters and were either canceled or delayed severely. To date, millions of software projects have been undertaken using any of thousands of software technologies, complicating research into topics that affect software-project outcomes. By concentrating on the extremes of possible results, I contend that the root causes of success and failure might be more clearly visible.

The overall results of my research have been published in several books cited in the reading list in the box on page 104, which contains a small sampling of the many studies available on software failures, successes, and risk factors. Here I concentrate on 10 of the worst current practices — those factors that most often lead to failure and disaster. I consider a software project a failure if it was:

- terminated because of cost or schedule overruns,
- experienced schedule or cost overruns in excess of 50 percent of initial estimates, or
- resulted in client lawsuits for contractual noncompliance.

The following paragraphs describe, in descending order, the 10 practices that contribute most to such software failures.

PRACTICE 1: *No historical software-measurement data.* Why should a lack of historical measurement data be the key to almost every software failure in the world? Because a lack of solid historical data makes project managers, executives, and clients blind to the realities of software development.

A lack of solid historical data makes project managers, executives, and clients blind to the realities of software development.

Suppose you are managing a type of software project that no company has ever built in less than 36 calendar months. As a responsible manager, you develop a careful estimate and critical-path analysis, then tell the client and your own executives that you think the project will require from 36 to 38 months to complete.

What often follows is an arbitrary rejection of your plan and a directive by either the client or your own executives to finish the project in *18* months. Where does this schedule come from? It is probably an arbitrary number totally unrelated to the size and complexity of the project or to the skill and experience of the development team. Once management creates such an arbitrary schedule, the project in question will usually be a disaster; it will certainly run late. From the day the directive is issued, the project is essentially doomed.

PRACTICE 2: *Rejection of accurate estimates.* The fundamental reason for many software disasters

0740-7459/96/$05.00 © 1996 IEEE

is that our industry lacks a solid empirical foundation of measured results. Thus, almost every major software project is subject to arbitrary and sometimes irrational schedule and cost constraints. Therefore, lack of accurate measurement data is the root cause for practices three through 10 and contributes to a host of secondary problems, including but not limited to:

- inability to perform return-on-investment calculations,
- susceptibility to false claims by tool and method vendors, and
- software contracts that are ambiguous and difficult to monitor.

PRACTICES 3 AND 4: *Failure to use automated estimating tools and automated planning tools.* Many factors must be dealt with when constructing an accurate software-cost estimate and developing a realistic project-development plan. Manual methods for estimating and planning are inadequate for large systems. There are about 50 commercial software-cost estimating tools and more than 100 project-planning tools on the market. Software projects that use such tools concurrently have a much greater probability of success than those that attempt to estimate and plan by manual means.

Some very common tool combinations include software-cost estimating tools such as Checkpoint, Cocomo, Estimacs, Price-S, or Slim and general-purpose project management tools such as Microsoft Project, Primavera, Project Manager's Workbench, or Timeline; the two categories complement each other.

Estimating tools have a built-in knowledge base of specific software factors such as the impact of various methodologies and tools. The project-management tools can focus down to the level of individual employees. Together, the combination of estimating and planning tools leads to accurate and realistic outcomes not easily overridden by clients or executives.

PRACTICES 5 AND 6: *Excessive, irrational schedule pressure and creep in user requirements.* These practices also derive from a lack of solid empirical data. Once management or the client imposes an arbitrary and irrational schedule, they insist on adhering to it, which results in shortcuts to design, specification, and quality control that damage the project beyond redemption.

At the same time, the original requirements for the project tend to grow continuously throughout the development cycle. The combination of continuous schedule pressure and continuous growth in unanticipated requirements results in a very hazardous pairing.

Software requirements change at an average rate of about 1 percent per calendar month. Thus, for a project with a 12-month schedule, more than 10 percent of the final delivery will not have been defined during the requirements phase. For a 36-month project, almost a third of the features and functions may have been added as afterthoughts.

Manual methods for estimating and planning are inadequate for large systems.

These are only average results. I have observed a three-year project in which the delivered product exceeded the functions in the initial requirements by about 289 percent. Fortunately, the function-point metric now lets project teams directly measure the rate at which requirements creep or grow.

PRACTICES 7 AND 8: *Failure to monitor progress and to perform formal risk management.* These two practices often occur together and are at least strongly associated. To date, no standard checkpoints for software projects exist that function as clear and unambiguous indicators of possible failure or success. This lack leads to surrogates, such as the well-known but subjective "90 percent completion" assertions by project managers or technical personnel.

Nor is there a standard checklist of software-risk factors that should be evaluated. Even a rudimentary checklist of software-risk control factors would be helpful:

- What measurement data from similar projects has been analyzed?
- What measurement data on this project will be collected?
- Have formal estimates and plans been prepared?
- How will creeping user requirements be handled?
- What milestones will be used to indicate satisfactory progress?
- What series of reviews, inspections, and tests will be used?

PRACTICES 9 AND 10: *Failure to use design reviews and code inspections.* Sadly, most projects that end in disaster might not if their development teams used one of the most effective technologies in all of software engineering: Formal design and code inspections. These two practices have a 30-year history of successful deployment on large and complex software systems. All "best-in-class" software producers use software inspections. The measured defect-removal efficiency of inspections is about twice that of most forms of software testing: about 60 percent for inspections versus 30 percent for most kinds of testing.

MINIMIZING RISKS. Actually, practices 6 through 10 are intertwined and have a common solution. A well-formed risk-analysis and milestone-tracking program for software projects depends, quite simply, on successful completion of formal design and code inspections.

Overcoming the risks I've shown here is largely a matter of opposites, or of doing the reverse of what the risk indicates. Thus, a well-formed software project will create accurate estimates derived from empirical data and supported by automated tools for handling the critical-path issues. Such estimates will be based on the actual capabilities of the development team, and will not be

arbitrary creations derived without any empirical data.

Neither executives nor clients should reject well-formed estimates just because they don't like the results. Unless there is solid, empirical evidence that the project can be done for a lower cost or in a shorter time span, arbitrarily overriding a formal estimate and development plan courts disaster.

Creeping requirements can be minimized by approaches such as joint application design or prototyping. Risk analysis and quality control will be major aspects of the software team's responsibility. Milestones for completing the project should include the successful completion of formal inspections for at least five deliverables: requirements, specifications, source code, test materials, and user manuals.

Software failures are caused primarily by errors and poor judgment on the part of managers, executives, and clients — not errors made by the technical teams. The root cause of these failures is the lack of accurate measurement data, which blinds management and clients to what is possible and what might be impossible.

AVOIDING DISASTER. Successful software projects can result from avoiding the more serious mistakes that lead to disaster. Specifically, we must

- look at the actual results of similar projects;
- make planning and estimating formal activities;
- plan for and control creeping requirements;
- use formal inspections as milestones for tracking project progress; and
- collect accurate measurement data, during the current project, to use with future projects.

There is no substitute for solid empirical data used by capable project managers who are supported by automated estimating and planning tools. This combination can almost always be successful. By contrast, no data at all, unprepared managers, and manual estimating and planning are consistently characteristic of our industry's major software disasters. ◆

Capers Jones is chairman of Software Productivity Research, Inc., an international management-consulting company and a developer of software project-management tools located in Burlington, Massachusetts.

SUGGESTED READINGS ON SOFTWARE RISK AVOIDANCE

The following books are not the only ones on software risks, but they cover the essential topics.

- F. Brooks, *The Mythical Man Month*, Addison-Wesley, Reading, MA, 1995, 295 pp.: This is the 20th anniversary edition of a software classic. Initially published in 1975, Fred Brooks' thoughtful historical analysis examined why software is so often delivered late. The 20th-anniversary edition adds new material and lets Brooks explore recent changes in software technologies.
- R.N. Charette, *Software Engineering Risk Analysis and Management*, McGraw Hill, New York, 1989; 325 pp.: Robert Charette is a pioneer in the exploration of software risk management. All of his books are useful. This is a very good introduction to the overall topic of software risk analysis.
- R. N. Charette, *Applications Strategies for Risk Analysis*, McGraw Hill, New York, 1990, 570 pp.: This large book is a more complete coverage of risk-related topics than Charette's other book. Both are useful for software project managers, and recommended.
- T. DeMarco and T.Lister, *Peopleware*, Dorset House, New York, 1987, 200 pp.: This book was one of the first to deal with the social and even ergonomic topics that affect the outcomes of software projects.
- T. Gilb and D. Graham, *Software Inspections*, Addison-Wesley, Reading, MA, 1993, 471 pp.: Although formal software inspections were invented in the 1960s by Michael Fagan and colleagues at IBM's programming laboratory in Kingston, NY, Tom Gilb has become one of the most enthusiastic supporters of the concept.
- S. Grey, *Practical Risk Assessment for Project Management*, John Wiley & Sons, New York, 1995, 140 pp.: Managers seldom have the time or inclination to absorb the full-scale risk literature embodied in Charrette's or Jones' 500-700-page books. This small, 140-page book provides an introduction to the topic of risk analysis.
- C. Jones, *Assessment and Control of Software Risks*, Prentice-Hall, Reading, Mass., 1994, 711 pp.: This book covers some 65 technical and sociological risk factors associated with software development and maintenance operations. The data has been collected during the course of SPR's software-process assessment activities. The book includes quantitative data on "best-in-class" quality and productivity results derived from the top 10 percent of SPR's clients.
- C. Jones, *Patterns of Software System Failure and Success*, International Thomson, Boston, Mass., 1996, 250 pp.: This book provides an analysis of software projects larger than 5000 function points that occupy the ends of the effectiveness spectrum: They were either disasters or set new records for quality and productivity. On the whole, management problems appear to outweigh technical problems in both the successes and failures.