Why Big Software Projects Fail: The 12 Key Questions

Watts S. Humphrey
The Software Engineering Institute

In spite of the improvements in software project management over the last several years, software projects still fail distressingly often, and the largest projects fail most often. This article explores the reasons for these failures and reviews the questions to consider in improving your organization's performance with large-scale software projects. Not surprisingly, considering these same questions will help you improve almost any large or small project with substantial software content. The principal questions concern why large software projects are hard to manage, the kinds of management systems needed, and the actions required to implement such systems. In closing, the author cites the experiences of projects that have used the methods described and cites sources for further information on introducing the required practices.

Software project failures are common, and the biggest projects fail most often. There are always many excuses for these failures, but there are a few common symptoms. Some years ago, before the invention of the Capability Maturity Model® (CMM®) and CMM Integration℠ (CMMI®) the principal problem was the lack of plans [1, 2]. In the early years, I never saw a failed project that had a plan, and very few unplanned projects were successful.

The methods defined for CMM and CMMI Levels 2 and 3 helped to address this problem. As the Standish data in Figure 1 shows, the success rate for software organizations improved between 1994 and 2000, and much of this improvement was due to more widespread use of sound project management practices [3]. Still, with less than 30 percent of our projects successful, those of us who are software professionals have little to be proud of.

The definition of a successful project is one that completed within 10 percent or so of its committed cost and schedule and delivered all of its intended functions. Challenged projects are ones that were seriously late or over costs or had reduced functions. Failed projects never delivered anything. Figure 2 (see page 26) shows another cut of the Standish data by project size. When looked at this way, half of the smallest projects succeeded, while none of the largest projects did. Since large projects still do not succeed even with all of the project management improvements of the last several years, one begins to wonder if large-scale software projects are inherently unmanageable.

Question 1: Are All Large Software Projects Unmanageable?

There are some large, unprecedented projects that are so risky that they would likely be challenged under almost any management system. But some large projects have succeeded. Two examples are the Command Center Processing and Display System Replacement (CCPDS-R) project, described by Walker Royce, and the operating system (OS)/360 project in my former group at IBM [4, 5]. The CCPDS-R was a U.S. Air Force installation at Cheyenne Mountain in Colorado. It had about 100 developers at its peak. The OS/360 was the operating system to support the IBM 360 line of computers, and included the control program, data management, languages, and support utilities. Its development team consisted of about 3,000 software professionals.

Both of these projects placed heavy emphasis on planning, and both adopted an evolutionary development strategy with multiple releases and phased specifications. Both projects also took a somewhat unconventional approach to motivating team member performance. For CCPDS-R, management distributed 50 percent of the project award fee to the development team members. This built their loyalty and commitment to success, and maintained team motivation throughout the job. The CCPDS-R project was delivered on schedule and within contracted costs.

By the time I took over the OS/360 project some years ago, we had all learned that the proper strategy for building big software-intensive systems was to break the job into as many small incremental releases as practical. Since this strategy required organization-wide coordination, our very first action was to have all the development teams in all the involved laboratories produce their own plans and coordinate them through a central build-and-release group. Then, we based the company's commitments on the dates that the teams provided. In no case did IBM commit to any date that was not supported by a plan that had been developed by the team that was to do the work.

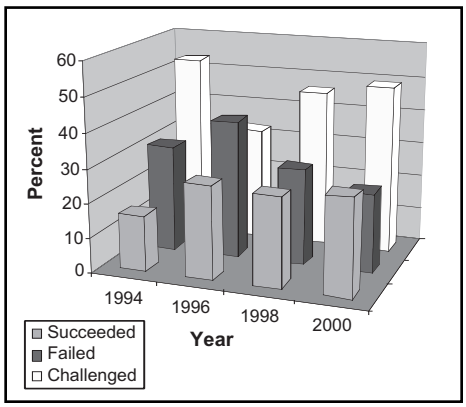
These plans extended through 19 releases over a period of 30 months. Most importantly, they provided the focus we all needed to coordinate the work of 15 laboratories in six countries and to promptly recognize and address the myriad problems that inevitably arose. The developers were personally committed to their schedules, and they delivered every one of these releases on or ahead of the committed schedules. So, at least based on this limited sample, some large software projects can be managed successfully. However, because the success rate is so low, large-scale software projects remain a major project management challenge.

Question 2: Why Are Large Software Projects Hard to Manage?

While large software projects are undoubtedly hard to manage, the key question is "Why?"

Historically, the first large-scale management systems were developed to manage armies. They were highly autocratic, with the leader giving orders and the

Figure 1: *Project Success History [3]*



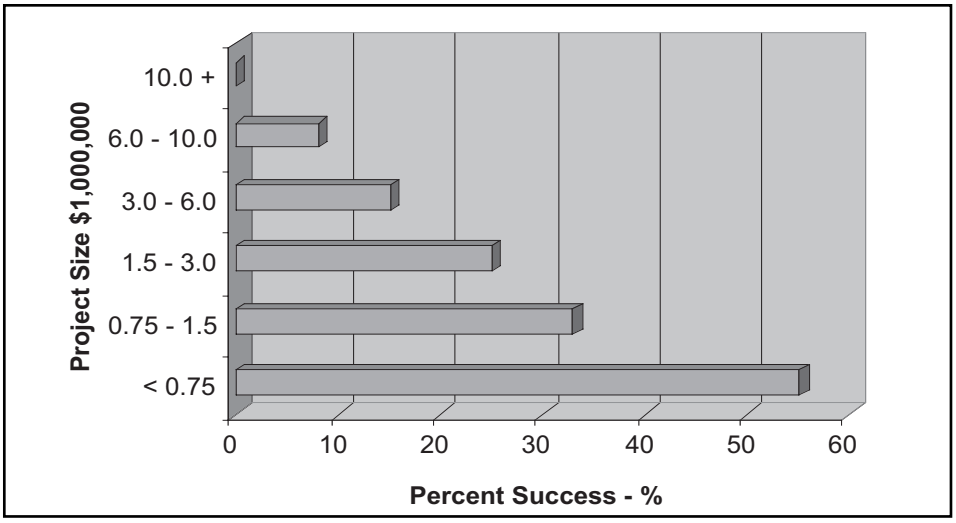


Figure 2: *Success Rate by Project Size [3]*

troops following. Over time, work groups were formed for major construction projects such as temples, palaces, fortifications, and roads. The laborers were mostly slaves, and again, the management system was highly autocratic. The workers did what they were told or they were punished.

This army-like structure was essentially the only management system for many years until the Greek city-states introduced democratic political systems. However, these democratic principles were primarily used for governing, not for project management. Somewhat later, a totally different management system was used to build cathedrals. This work was largely done by volunteer artisans who managed themselves under the guidance of a master builder. Since building a cathedral often took 50 years or more, the cathedral-management system is not a good model for modern large-scale software projects. However, it did produce some beautiful results. This cathedral-building management system was not used for anything but cathedrals for many years, but it has recently had some success as the guiding principle for the open source software development community [6].

The first large-scale management systems were developed for armies. Leaders gave orders and troops followed. With training and discipline, this approach could work even amid chaos and confusion.

Large-scale management systems were eventually applied to major construction projects. The system was highly autocratic; workers did what they were told or they were punished.

The next major management innovation was the factory. Factories started producing clothing and were soon used for making all kinds of goods. Again, however, the factory management system was autocratic, with management directing and workers doing. While the factory model improved productivity, it was not without its problems. The early work of Frederick Winslow Taylor about 100 years ago and the more recent work of W.E. Deming, J.M. Juran, and others has improved the effectiveness of this model by redefining the role of the worker. The modern view is that to do quality work for predictable costs and schedules, workers must be treated as thinking and feeling participants rather than merely as unfeeling drudges. However, to date, these methods have had limited application to software [7, 8, 9].

The factory/army system has persisted and now characterizes the modern corporate structure where senior management decides and everybody else follows. Many managers would contend that they listen to their people while making decisions. However, employees generally view corporate management as autocratic and few feel that they could influence a senior manager's decisions. Some managers even argue that autocratic management is the only efficient style for running large projects and organizations. Democratic debates would take too long and decisions would not be made by the most important or knowledgeable people.

Regardless of the validity of this view, the hierarchical management style does not work well for managing large software projects. Unfortunately, except for the cathedral-building system, there is no other proven way to manage large-scale work. So, if we want to have successful large-scale software projects, we must develop a project management system that is designed for this purpose.

Question 3: Why Is Autocratic Management Ineffective for Software?

Before developing a new management system, we should first understand why the current one does not work. To answer this question, we must explore the nature of software work and how it differs from other, more manageable work. Software and software-like work have characteristics that are particularly difficult to manage. From a management perspective, the principal difference between managing traditional hardware projects and modern software work concerns management visibility.

With manufacturing, armies, and traditional hardware development, the managers can walk through the shop, battlefield, or lab and see what everybody is doing. If someone is doing something wrong or otherwise being unproductive, the manager can tell by watching for a few minutes. However, with a team of software developers, you cannot tell what they are doing by merely watching. You must ask them or carefully examine what they have produced. It takes a pretty alert and knowledgeable manager to tell what software developers are doing. If you tell them to do something else or to adopt a new practice, you have no easy way to tell if they are actually working the way you told them to work.

Some might argue that hardware work is not actually that different from software work and that, at least for some hardware tasks and most system engineering jobs, the work is equally opaque to management. This is certainly true, particularly when the hardware engineers are producing microcode, using hardware design languages, or working with simulation or layout tools. Today, as modern technical specialties increasingly overlap, many hard-

ware projects now share the same characteristics as large software projects. When hardware development and system-engineering work have the characteristics of software work, they should be managed like software. However, since these systems groups generally tend to be relatively small, they do not yet present the same project-manageability problems as large-scale software.

Question 4: Why Is Management Visibility a Problem for Software?

Since most software developers are dedicated and hard-working professionals, why is management visibility a problem?

The problem is that the manager cannot tell where the project stands. To manage modern large-scale technical work, you must know where the project stands, how rapidly the work is being done, and the quality of the products being produced. With earlier hardware-development projects, all of this information was more-or-less visible to the manager, while with modern software and systems projects it often is not.

This is a problem because large development projects, whether hardware or software, always run into problems, and every problem involves more work. While developers can invariably overcome small problems, every problem adds to the workload and delays the job. Each little slip is generally manageable by itself, but over time, problems add up, and sooner or later the project is in serious trouble.

The project manager's job is to identify these small daily slips and to take steps to counter them. As Fred Brooks said, "Projects slip a day at a time" [10]. With traditional hardware projects, the manager could usually see these one- and two-day slips and could do something about them. With modern, complex, software-intensive systems, the daily schedule slips are largely invisible. So, with large-scale software work, the managers generally do not see the schedule problem until it is so big that it is obvious. Then, however, it is usually too late to do much about it.

Question 5: Why Can't Managers Just Ask the Developers?

If the managers cannot see where the developers stand, why not just ask them?

Most developers would be glad to tell their managers where they stood on the job. The problem is that, with current software practices, the developers do not know where they stand any more than the managers do. The developers know what they are doing, but they do not have personal plans, they do not measure their work, and they do not track their progress. Without these practices to guide them, software people do not know with any precision where they are in the job. They could tell the manager that they are pretty close to schedule or 90 percent done with coding, but the fact is that they do not really know. Again, as Brooks said, "...programmers generally think that they are 90 percent through with the coding for more than half of the project" [10].

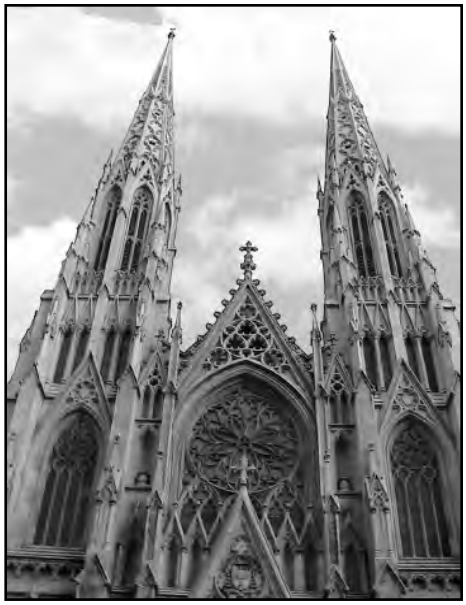
Unless developers plan and track their personal work, that work will be unpredictable. Furthermore, if the cost and schedule of the developers' personal work is unpredictable, the cost and schedule of their teams' work will also be unpredictable. And, of course, when a project team's work is unpredictable, the entire project is unpredictable. In short, as long as individual developers do not plan and track their personal work, their projects will be uncontrollable and unmanageable.

Anyone who has managed software development will likely argue that this is an overstatement. Although you may not know precisely where each developer's work stands, you can usually get a general idea. Since about a third to a half of the small projects are successful when the developers do not plan and track their personal work, such projects can be managed. So why should the lack of sound personal software practices be a problem for large projects?

It is true that software projects are not totally unmanageable. As Figures 1 and 2 show, the worst problem is with the very large software projects. On small projects, some uncertainty about each team member's status is tolerable. However, as projects get bigger and communications lines extend, precise status information becomes more important. Without hard data on project status, people communicate opinions, and their opinions can be biased or even wrong. When filtered through just a few layers of management, imprecise project status reports become so garbled that they provide little or no useful information. Then these large-scale software projects end up being run with essentially no management visibility into their true status, issues, and problems.

Question 6: Why Do Planned Projects Fail?

Today, with CMM and CMMI, most large software projects are planned, and they use methods like Program and Evaluation and Review Technique (PERT) and earned value to track progress. Why is that not adequate?



Work on cathedrals was done by volunteer artisans who managed themselves under the guidance of a master builder. This approach has had some success in open source software development.

The problem is with the imprecision and inaccuracy of most software project plans. Most projects have major milestones such as specifications complete, design complete, code complete, and the like. The problem is that on real software projects, few of these high-level tasks have crisp completion dates. The requirements work generally continues throughout design and even into implementation and test; coding usually starts well before design completion and continues through most of testing.

A few years ago, the management of a large software organization asked me to review their largest project. They told me that the code completion milestone had already been met on schedule. However, I found that very little code had actually been released to test. When I met with the development teams, they did not know how much code they had written or what remained to be done. It took a full week to get a preliminary count, and it was a month before we got accurate data. It was another 10 months before all of the coding was actually completed. It is not that developers lie, just that without objective data, they have no way to know precisely where they stand. When they are under heavy schedule pressure, people try to respond. Since we all know that the bearer of bad news tends to be blamed, no one dares to question the schedule and everyone gives the most optimistic story they can.

Question 7: Why Not Just Insist on Detailed Plans?

Why cannot management just insist on more detailed plans? Then they could have more precise measures of project status.

While this would seem reasonable, the issue is, "Whose plans are they?" Detailed plans define precisely how the work is to be done. When the managers make the plans, we have the modern-day equivalent of laborers building pyramids. The managers tell the workers what to do and how to do it, and the workers presumably do as they are told.

While this has been the traditional approach for managing labor, it has become progressively less effective for managing high-technology work, particularly software. The principal reason is that the managers do not know enough about the work to make detailed plans. That is why many of these software-intensive projects typically have very generalized plans. This provides the developers with the flexibility they need to do creative work in the way that they want to. The current system is therefore the modern equivalent of the cathedral-building system where the developers act like artisans. The unfortunate consequence is that, without Herculean effort, it often seems that the natural schedule for such projects could easily approach 50 years.

Question 8: Why Not Tell the Developers to Plan Their Work?

The obvious next step would be to tell the developers to make their own detailed plans. Why would this not work?

There are three problems. First, most developers do not want to make plans; they would rather write programs. They view planning as a management responsibility. Second, if you told them to make plans, they would not know how to do it. Few of them have the skill and experience to make accurate or complete plans. Finally, making accurate, complete, and detailed plans means that the developers must be empowered to define their own processes, methods, and schedules. Few managers today would be willing to cede these responsibilities to the software developers, at least not until they had evidence that the developers could produce acceptable results.

Question 9: How Can We Get Developers to Make Good Plans?

It seems that the problem of effectively managing large software projects boils down to two questions: How can we get the software developers and their teams to properly make and faithfully follow detailed plans, and how can we convince management to trust the developers to plan, track, and manage their own work?

To get the developers to make and follow sound personal plans, you must do three things: provide them with the skills to make accurate plans, convince them to make these plans, and support and guide them while they do it.

Providing the skills is just a question of training. However, once the developers have learned how to make accurate plans and to measure and track their work against these plans, they usually see the benefits of planning and are motivated to plan and track their own and their team's work. So, it is possible that developers *can* be taught to plan and, once they learn how, they are generally willing to make and follow plans [11].

Question 10: How Can Management Trust Developers to Make Plans?

This is the biggest risk of all: Can you trust developers to produce their own plans and to strive for schedules that will meet your objectives?

This question gets to the root of the problem with autocratic management methods: trust. If you trust and empower your software and other high-technology professionals to manage themselves, they will do extraordinary work. However, it cannot be blind trust. You must ensure that they know how to manage their own work, and you must monitor their work to ensure that they do it properly. The proper monitoring attitude is not to be distrustful, but instead, to show interest in their work. If you do not trust your people, you will not get their whole-hearted effort and you will not capitalize on the enormous creative potential of cohesive and motivated teamwork. It takes a leap of faith to trust your people, but the results are worth the risk.

Question 11: What Are the Risks of Changing?

Every change involves some risk. However, there is also a cost for doing nothing. If you are happy with how your large software projects are performing, there is no need to change. However, few managers or professionals are comfortable with the current state of software practice, particularly for large-scale projects. So, there are risks to changing and risks to not changing. The management challenge is to balance these risks before deciding what to do.

There are two risks to changing to a new management system for large-scale software projects. First, it costs time and money to train the developers to plan and track their work and to train the managers to use a new management system. Then comes the risk of using these methods on a real project. While you will see some early benefits, you will not know for sure whether this new management system is truly effective for you until the first project is completed and you can analyze the results.

This brings up a related and even more difficult problem: On large multi-year projects, there is not time to run pilots. You must pick a management strategy and go with it. However, since almost all large software-intensive projects are now failing anyway, the biggest risk is *not changing*. Perhaps the biggest shock for most managers is realizing that they are part of the problem, and that they have to change their behavior to get the kind of large-system results they want.

These problems are common to all change efforts. The way to manage these problems is to examine the experiences of others and to minimize your exposure by carefully planning your change effort and getting help from people who have already used the methods you plan to introduce. Of course the alternative is to hope that things will get better without any changes. With this choice, however, your large-systems projects will almost certainly continue to perform much as they have in the past.

Question 12: What Has Been the Experience So Far?

The Software Engineering Institute (SEI℠) has developed a method called the Team Software Process℠ (TSP℠) that follows the concepts described in this article [11]. With the TSP[1], if you properly train and support your development people and if you follow the SEI's TSP introduction strategy, your teams will be motivated to do the job properly. The team members' personal practices will be defined, measured, and managed; team performance will also be defined, measured, and managed; and the project's status and progress will be precisely reported every week. Although this will not guarantee a successful project, these practices have worked for the several dozen projects that have tried them so far.

Moreover, there is one caveat. These

practices have proven effective for teams of up to about 100 members, as well as for teams composed of multiple hardware, systems, and software professionals. They have even worked for distributed teams from multiple geographic locations and organizations. Although these methods should scale up to very large projects, the TSP has not yet been tried with projects of over 100 professionals. I know from personal experience, however, that these practices will address many of the problems faced by the managers of software organizations of several thousand developers.

The other articles in this issue describe the TSP experiences of several organizations. They describe how these practices have worked on various kinds of projects and how they could help your organization.◆

Acknowledgements

Many people have participated in the work that led to this article, so I cannot thank them all personally. However, without their willingness to try new methods and to take the risks that always accompany change, this work would not have been possible. So, to everyone who participated in the early CMM and CMMI work and to all of those who have learned and used the Personal Software Process℠ and TSP, you have my profound gratitude. I have also had the advice and support of several people in writing this article. My special thanks go to Dan Burton, Noopur Davis, Bill Peterson, Marsha Pomeroy-Huff, and Walker Royce.

References

1. Humphrey, Watts S. Managing the Software Process. Reading, MA: Addison-Wesley, 1989.
2. Chrissis, Mary Beth, Mike Konrad, and Sandy Shrum. CMMI – Guidelines for Process Integration and Process Improvement. Reading, MA: Addison Wesley, 2003.
3. The Standish Group International, Inc. Extreme Chaos. The Standish Group International, Inc., 2001.
4. Royce, Walker. Software Project Management, A Unified Framework. Reading, MA: Addison-Wesley, 1998.
5. Humphrey, Watts S. "Reflections on a Software Life." In the Beginning, Recollections of Software Pioneers. Robert L. Glass, Ed. Los Alamitos, CA: IEEE Computer Society Press, 1998.
6. Raymond, Eric S. The Cathedral and the Bazaar. Cambridge, MA: O'Reilly Publishers, 1999.
7. Deming, W. Edwards. The New Economics for Industry, Government, Education. 2nd ed. The MIT Press, Cambridge, MA, 2000.
8. Juran, J.M., and Frank M. Gryna. Juran's Quality Control Handbook, Fourth Edition. New York: McGraw-Hill Book Company, 1988.
9. Taylor, Frederick Winslow. The Principles of Scientific Management. New York: Harper and Row, Publishers, Inc., 1911.
10. Frederick P. Brooks. The Mythical Man-Month. Reading, MA: Addison Wesley, 1995.
11. Humphrey, Watts S. Winning With Software: An Executive Strategy. Reading, MA: Addison-Wesley, 2002.

Note

1. The Software Engineering Institute offers courses and transition services to help organizations introduce the TSP. Additional information is available at <tsp@sei.cmu.edu> or at <www.sei.cmu.edu/tsp>.

About the Author

Watts S. Humphrey joined the Software Engineering Institute (SEI℠) of Carnegie Mellon University after his retirement from IBM in 1986. He established the SEI's Process Program and led development of the Software Capability Maturity Model®, the Personal Software Process℠, and the Team Software Process℠. During his 27 years with IBM, he managed all IBM's commercial software development and was vice president of Technical Development. He holds graduate degrees in physics and business administration. He is an SEI Fellow, an Association for Computing Machinery member, an Institute of Electrical and Electronics Engineers Fellow, and a past member of the Malcolm Baldrige National Quality Award Board of Examiners. He has published several books and articles and holds five patents.

Carnegie Mellon University
4500 Fifth AVE
Pittsburgh, PA 15213-2612
Phone: (941) 924-4169
Fax: (941) 925-1573
E-mail: watts@sei.cmu.edu

CROSSTALK
The Journal of Defense Software Engineering

Get Your Free Subscription

Fill out and send us this form.

OO-ALC/MASE
6022 FIR AVE
BLDG 1238
HILL AFB, UT 84056-5820
FAX: (801) 777-8069 DSN: 777-8069
PHONE: (801) 775-5555 DSN: 775-5555

Or request online at www.stsc.hill.af.mil

NAME:_____

RANK/GRADE:_____

POSITION/TITLE:_____

ORGANIZATION:_____

ADDRESS:_____

_____

BASE/CITY:_____

STATE:_____ZIP:_____

PHONE:(____)_____

FAX:(____)_____

E-MAIL:_____

CHECK BOX(ES) TO REQUEST BACK ISSUES:

OCT2003	☐	INFORMATION SHARING
NOV2003	☐	DEV. OF REAL-TIME SW
DEC2003	☐	MANAGEMENT BASICS
JAN2004	☐	INFO FROM SR. LEADERSHIP
MAR2004	☐	SW PROCESS IMPROVEMENT
APR2004	☐	ACQUISITION
MAY2004	☐	TECH.: PROTECTING AMER.
JUN2004	☐	ASSESSMENT AND CERT.
JULY2004	☐	TOP 5 PROJECTS
AUG2004	☐	SYSTEMS APPROACH
SEPT2004	☐	SOFTWARE EDGE
OCT2004	☐	PROJECT MANAGEMENT
NOV2004	☐	SOFTWARE TOOLBOX
DEC2004	☐	REUSE
JAN2005	☐	OPEN SOURCE SW
FEB2005	☐	RISK MANAGEMENT

TO REQUEST BACK ISSUES ON TOPICS NOT LISTED ABOVE, PLEASE CONTACT KAREN RASMUSSEN AT <STSC.CUSTOMERSERVICE@HILL.AF.MIL>.