Software projects are still late, over budget, and unpredictable. Sometimes the entire project fails before ever delivering an application. This clear, commonsense review of fundamental project management techniques reminds us that we still have a long way to go.

Critical Success Factors In Software Projects

John S. Reel, Trident Data Systems

Throughout the fifty-odd years of software development, the industry has gone through at least four generations of programming languages and three major development paradigms. We have held countless seminars on how to develop software correctly, forced many courses into undergraduate degree programs, and introduced standards in our organizations that require specific technologies. Still, we have not improved our ability to successfully, consistently move from idea to product. In fact, recent studies document that, while the failure rate for software development efforts has improved in recent years, the number of projects experiencing severe problems has risen almost 50 percent.[1] There is no magic in managing software development successfully, but a number of issues related to software development make it unique.

0740-7459/99/$10.00 © 1999 IEEE

Managing Complexity

Several characteristics of software-based endeavors complicate management. First, software-based systems are exceptionally complex. In fact, many agree that "the basic problem of computing is the mastery of complexity."[2] Because software developers must deal with complex problems, they are generally very intelligent and complex individuals, which also complicates the management formula. Add the fact that developers are trying to hit a moving target—user requirements—and you get a volatile mixture of management issues.

These and many other influences contribute to a fantastically high failure rate among software development projects. The Chaos study, published by the Standish Group, found that 26 percent of all software projects fail (down from 40 percent in 1997), but 46 percent experience cost and schedule overruns or significantly reduced functionality (up from 33 percent in 1997).[1] The study also shows that the completion rate has improved because companies have trended towards smaller, more manageable projects—not because the management techniques have improved. Can you imagine a construction firm completing only 74 percent of its buildings and completing only 54 percent of the buildings within schedule and budget? To change this trend, we must place special emphasis on certain factors of the management process.

You may think the answers lie in elaborate analysis methodologies, highly advanced configuration management techniques, or the perfect development language. Those elements of the technology landscape are as important as highly scientific and analytical research in analysis and design methodologies, project management, and software quality. However, blueprints of the latest train technology didn't improve life in the Wild West until rail companies invested in the fundamental aspects of train transportation—tracks and depots. Likewise in software, more "advanced" technologies are far less critical to improving practice than embracing what I believe are the five essential factors to managing a successful software project:

1. Start on the right foot.
2. Maintain momentum.
3. Track progress.
4. Make smart decisions.
5. Institutionalize post-mortem analyses.

Granted, even a detailed review of these may leave you wondering what's new here. Not much—this is common-sense, basic management stuff. And yet these principles are not commonly employed. If they were, we would not see such high failure rates.

Start on the Right Foot

It is difficult to call any of these factors most important, since they are all critical to the success of large development efforts. However, getting a project set up and started properly certainly leads this class of factors. Just as it is difficult to grow strong plants in weak soil, it is almost impossible to successfully lead a development effort that is set up improperly. Tom Field analyzed pitfalls in software development efforts and gave 10 signs of IS project failures—at least seven of which are fully determined before a design is developed or a line of code is written.[3] Therefore, 70 percent of the dooming acts occur before a build even starts.

At least seven of 10 signs of IS project failures are determined before a design is developed or a line of code is written.

Here are 10 signs of IS project failure:[3]

1. Project managers don't understand users' needs.
2. The project's scope is ill-defined.
3. Project changes are managed poorly.
4. The chosen technology changes.
5. Business needs change.
6. Deadlines are unrealistic.
7. Users are resistant.
8. Sponsorship is lost.
9. The project lacks people with appropriate skills.
10. Managers ignore best practices and lessons learned.

Given this information, what can we do to get projects off to a successful start?

Set realistic objectives and expectations—for everyone

The first objective in getting a project off to a good start is to get everyone on the same wavelength. Management, users, developers, and designers must all have realistic expectations. In

case your customers haven't heard, remind them routinely that this system will not solve all of their problems and it will probably create new issues. The new system should cost-effectively solve more problems than it creates. The developers must also understand that the customers do not know exactly what they want, how they want it, or how it will help them. Often, they don't even know how much they can spend. Everyone has to come to the table with their eyes open, willing to cooperate and listen. To avoid later heartache, pay strict attention to the commitments made by both sides.

By the time you figure out you have a quality problem, it is probably too late to fix it.

Build the right team

Next, you must put together the right team. First ensure that you have enough resources to get the job done. If you do not get commitments for resources up front, the effort is doomed. If management is not excited enough about the effort to give it enough resources, you may not have the support necessary for success. Remember, too, that you will likely need more resources than you think. We are all inherently optimistic, so guard your personnel projections and err on the high side from the start.

Building the right team means getting good people. This is hard because companies usually want to place personnel moving off other efforts. Sometimes these people are good resources, but not always. However, also recognize that you do not need, or want, all of the very best designers and developers. In my experience, staffing around 20 percent of the team with the best available works well. This figure is loosely supported in Fred Brooks' essay "The Surgical Team."[4] His team of about 10 people includes two who are real experts (the Chief Programmer and the Language Lawyer). Having too many stars creates ego issues and distractions, while not having enough can leave the team struggling with small problems.

The rest of the team should be good, solid developers with compatible personalities and work habits. The more advanced team members can step ahead into uncharted waters, develop the most critical algorithms and applications, and provide technical mentoring to the rest of the team.

The most critical element in selecting people is creating an environment in which they can excel, and that lets you focus more on technology than team dynamics. You don't want a team of clones, but you do want people who are compatible with one another and with the company and team environment you are striving to establish. For example, a married-with-kids, laid-back, nine-to-five developer might not work well on a team of young, single, forceful seven-to-eleven developers. This doesn't mean the former is any less qualified or productive. Actually, that laid-back developer may produce better code and be more productive than the rest of the group. If you think that first person brings a calming, focused influence without either "side" becoming overly frustrated, maybe it is a good fit after all. At any rate, you must take these factors into consideration when building your team.

Wherever possible, and it usually is possible, involve customers and users in the development. Not only does this help build higher levels of trust between developers and users, it also places domain experts within arm's reach of the developers throughout development. This increases the chance that you will develop a product that meets the user requirements.

Give the team what they think they need

Once you have built a strong team, you must next provide it with an environment that encourages productivity and minimizes distractions. First, do your best to arrange quiet, productive office spaces. This is often impossible given most corporate realities, but a comfortable office setting can yield dramatic results. Highly productive environments contain white boards, meeting areas (formal and informal), private office areas, and flexible, modern lab facilities. Add comfort elements such as stereos, light dimmers, coffee machines, and comfortable chairs; you will create an environment where people can focus on their work and forget the rest of the world.

Once you have a team with a productive office space, you need the proper equipment. Do not for any reason scrimp on equipment. The difference between state-of-the-art machines and adequate development systems is less than $1,000. You will probably spend at least $100,000 per year to keep a good developer, including salary, bonuses, benefits, training, and other related expenses. That extra $1,000 amortized over two years represents less

than 1/2 percent of the employee cost.

Finally, your team needs tools. Get good, proven tools from stable companies. Nothing will derail a project faster than using unsupported tools. The team also needs training on those tools; losing files and folders from ignorance and inexperience is painful and costly. The term tool does not just mean compiler. You also need analysis and design, configuration management, testing, back-up management, document production, graphics manipulation, and troubleshooting tools. This is, however, an area where going first-class does not necessarily mean spending the most money. Shop carefully, review a lot of options, and involve the entire team in the decision.

Maintain the Momentum

By now, you have your development team energized with strong co-workers, a great working environment, and some high-end hardware. Congratulations, you have momentum. The next critical factor is maintaining and increasing this momentum. Building momentum initially is easy, but rebuilding it is dreadfully difficult. Momentum changes often during the course of a development effort. These changes add up quickly, so it is crucial to quickly offset the negative shifts with positive ones.

You should focus on three key items to maintain or rebuild team momentum:

- Attrition—keep it low.
- Quality—monitor it early on and establish an expectation of excellence.
- Management—manage the product more than the people.

Attrition

Attrition is a constant problem in the software industry. It can spell disaster for a mid-stream software project, because replacement personnel must quickly get up to speed on software that is not complete, not tested, and probably not well-documented yet. A tremendous amount of knowledge walks out the door with the person leaving, and those left behind have a scapegoat for every problem from then on. Also, in this tight labor market, the lag time between when a person quits and when a replacement is hired can wreak havoc with even the most pessimistic schedules.

Quality

You cannot go back and add quality. By the time you figure out you have a quality problem, it is probably too late to fix it. Establish procedures and expectations for high levels of quality before any other development begins and hire developers proven to develop high-quality code. Have the developers participate in regular peer-level code reviews and external reviews.

Invariably, when a project is cruising along, everybody is excited, the status reports look great, and the GUI is awesome, everything goes wrong. There may be a bad test report, a failed demo, or a small change request from the customer that becomes the pebble that starts an avalanche. You fix one bug, or make one change, and cause two more. Suddenly, the development team that was making fantastic progress is mired in repairing and modifying code that has been in the bank for months.

Management

Manage your product more than your personnel. After all, the product is what you are selling. So, if your corporate culture can handle it, don't worry about dress codes or fixed work hours. Relax and let people deliver things at the last minute. Then critique their products. If the products are not acceptable, you can start working with the individuals to improve their products. The goal here is to not make individual issues team problems. Just because one or two people like to come in at 10:00 a.m. and work until 5:00 p.m., abusing the flexibility you give, doesn't mean you should dampen the environment for the whole team. Too often, project leads avoid confronting the individuals and merely "fix" the problem by setting arbitrary team rules. Soon, everyone is griping about deviant co-workers and the strict management. Those are the sounds of momentum slipping away. Roll a few of these decisions together, and the team is soon focused more on avoiding the rules or tattling on offenders than on producing a quality product.

Project leaders often avoid confronting individuals and merely "fix" a problem by setting arbitrary team rules.

When you do have a legitimate personnel problem, deal with it quickly. If you must let someone go, do it quickly and then meet with your team to explain what happened. As long as you are being

fair, these experiences will contribute to the team's cohesiveness and allow them to rebuild momentum quickly.

Track Progress

Consider the intangible nature of software compared to traditional brick-and-mortar construction. Construction results in a physical manifestation of a conceptual model—the blueprint becomes a building that people can touch and see. They can also touch and see all of the little pieces as they are being nailed, welded, glued, or screwed to the framework during construction. Software development begins as a conceptual model and results in an application, so there is no physical manifestation of software that can be touched and measured, especially during construction.

If you don't take time to figure out what happened during a project, both the good and the bad, you're doomed to repeat it.

A large problem in managing software development is figuring where you are in your schedule. How complete is a module? How long will it take to finish modules X, Y, and Z? These are hard questions to answer, but they must be addressed. If you don't know where you are in relation to the schedule, you cannot adjust and tweak to bring things back on track. Many methodologies exist for tracking progress; select one at the right level of detail for your effort, and use it religiously.

Make Smart Decisions

Making smart decisions often separates successful project leaders from failures. It shouldn't be hard to identify a bad decision before you make it. Choosing to rewrite a few of Microsoft's dynamically linked libraries to accommodate your design choices, for example, is a poor decision. Yet I have seen at least four major projects attempt such insanity. If your application needs to communicate across a serial connection, do you buy a commercial library of communications routines or develop your own from scratch? If you build it from scratch, you can then implement your own personally designed protocol. Bad call. Always use commercial libraries when available, and never try to create a new communications protocol. At best, it will cost you a fortune. At worst, it will sink your project.

People also consistently make bad decisions in selecting technologies. For example, how many people chose to develop applications for the Next platform? Most never finished their applications before that platform went away. When you pick a fundamental technology, whether a database engine, operating system, or communications protocol, you must do a business and a technical analysis. If the technology isn't catching market share and if a healthy company doesn't support it, you are building your project on a sandy foundation.

Because your foresight is fallible, use your design to insulate yourself from the underlying technology. Encapsulate the interface to new or niche technologies as much as possible. Think about which technologies will be prone to change over your product's lifetime and design your application to insulate—to a practical level—your code from those changes.

You will have many opportunities to make good decisions as you negotiate the customer's requirements. Strive to move the requirements from the complicated, "never been done before" category to the "been there, done that" category. Often, users request things that are marginally valuable without understanding the complexity. Explain the ramifications of complicated requirements and requirements changes in terms of cost and schedule. Help them help you.

Post-Mortem Analysis

Few companies institutionalize a process for learning from their mistakes. If you do not take time to figure out what happened during a project, both the good and the bad, you are doomed to repeat it.

What can you learn from a post-mortem analysis? First, you learn why your schedule estimates were off. Compensating for those factors in the next project will dramatically improve your estimating techniques. A post-mortem will also help you develop a profile for how your team and company develop software systems. Most companies and teams have personalities that strongly impact the development cycle. As you go through post-mortem

analyses, these personalities emerge as patterns rather than as isolated incidents. Knowing the patterns allows you to circumvent or at least schedule for them on your next project.

In his book *Managing Software Development Projects*, Neal Whitten offers six steps for executing a post-mortem review of a project:[5]

- Declare the intent: Announce at the beginning of the project that you will hold the review. Also define what topics will be addressed, and set the procedures.
- Select the participants: Choose representatives from each major group associated with the project. To ensure an objective review, management should not participate directly.
- Prepare the review: After the project is complete, assign review participants to gather data. This should include metrics, staffing, inter- and intragroup communications, quality, and process.
- Conduct the review: The actual review should not require more than a few days of meetings. All participants should start by presenting their findings and experiences with the project. Next, the group prepares two lists: things that went right and things that went wrong. Participants can then begin to work on what went wrong to develop solutions.
- Present the results: The participants should present the results to the development team and executive leadership.
- Adopt the recommendations: The company must implement the recommendations on upcoming projects. Without this follow-through, the process yields a marginal benefit.

The premise and benefit of performing post-mortem analyses are validated by the process improvement movement inspired by W. Edwards Deming during the late 1980s and early 1990s. He suggests objectively measuring a given process and using those measurements to evaluate the influence of changes to the processes. Only by measuring a system and analyzing those incremental measurements can you truly improve the system.[6]

Guess what? Your company's software methods and habits for developing software constitute a system. It is far less defined than an assembly line, but it is still a system. The post-mortem analysis allows you to modify that system for the next "production run."

These five critical factors hold true regardless of the design and development methodology, the implementation language, or the application domain. However, this is not an exhaustive list—many other factors influence the successful management of a software development effort. But if you master these five, you greatly increase the odds of completing your project on time and within budget. Just as important, you increase your chances of actually delivering something your users want. ❖

References

1. R. Whiting, "News Front: Development in Disarray," *Software Magazine*, Sept. 1998, p. 20.
2. J. Martin and C. McClure, *Structured Techniques for Computing*, Prentice Hall, Upper Saddle River, N.J.,1988.
3. T. Field, "When BAD Things Happen to GOOD Projects," *CIO*, 15 Oct. 1997, pp. 55-62.
4. F.P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison Wesley Longman, Reading, Mass., 1995.
5. N. Whitten, *Managing Software Development Projects*, John Wiley & Sons, New York, 1995.
6. R. Aguayo, *Dr. Deming: The American Who Taught the Japanese About Quality*, Fireside Books, New York, 1990.

About the Author

John S. Reel is the chief technology officer of Trident Data Systems, an information protection and computer networking company. He is a co-inventor of patented COMSEC technology. He received a BS in computer science from the University of Texas at Tyler and a PhD in computer science from Century University. He also worked for the US Department of Defense in systems support, software development, and management.

He is a member of the IEEE, the IEEE Computer Society, Information Systems Security Association, and the Armed Forces Communications and Electronics Association.

Readers may contact Reel at 6615 Gin Road, Marion, Texas 78124; e-mail jreel@tds.com.