Classic Mistakes Enumerated

Some ineffective development practices have been chosen so often, by so many people, with such predictable, bad results that they deserve to be called "classic mistakes." Most of the mistakes have a seductive appeal. Do you need to rescue a project that's behind schedule? Add more people! Do you want to reduce your schedule? Schedule more aggressively! Is one of your key contributors aggravating the rest of the team? Wait until the end of the project to fire him! Do you have a rush project to complete? Take whatever developers are available right now and get started as soon as possible!

Developers, managers, and customers usually have good reasons for making the decisions they do, and the seductive appeal of the classic mistakes is part of the reason these mistakes have been made so often. But because they have been made so many times, their consequences have become easy to predict, and they rarely produce the results that people hope for.

This section enumerates three dozen classic mistakes. I have personally seen each of these mistakes made at least once, and I've made many of them myself. You'll recognize many of them from Case Study 3-1.

The common denominator in this list is that you won't necessarily get rapid development if you avoid the mistake, but you will definitely get slow development if you don't avoid it.

If some of these mistakes sound familiar, take heart. Many other people have made the same mistakes, and once you understand their effect on development speed you can use this list to help with your project planning and risk management.

Some of the more significant mistakes are discussed in their own sections in other parts of this book. Others are not discussed further. For ease of reference, the list has been divided along the development-speed dimensions of people, process, product, and technology.

People

Here are some of the people-related classic mistakes.

#1: Undermined motivation. Study after study has shown that motivation probably has a larger effect on productivity and quality than any other factor (Boehm 1981). In Case Study 3-1, management took steps that undermined morale throughout the project--from giving a hokey pep talk at the beginning to requiring overtime in the middle and going on a long vacation while the team worked through the holidays to providing bonuses that work out to less than a dollar per overtime hour at the end.

#2: Weak personnel. After motivation, either the individual capabilities of the team members or their relationship as a team probably has the greatest influence on productivity (Boehm 1981, Lakhanpal 1993). Hiring from the bottom of the barrel will threaten a rapid development effort. In the case study, personnel selections were made with an eye toward who could be hired fastest instead of who would get the most work

done over the life of the project. That practice gets the project off to a quick start but doesn't set it up for rapid completion.

#3: Uncontrolled problem employees. Failure to deal with problem personnel also threatens development speed. This is a common problem and has been well-understood at least since Gerald Weinberg published *Psychology of Computer Programming* in 1971. Failure to take action to deal with a problem employee is the most common complaint that team members have about their leaders (Larson and LaFasto 1989). In Case Study 3-1, the team knew that Chip was a bad apple, but the team lead didn't do anything about it. The result--redoing all of Chip's work--was predictable.

#4: Heroics. Some software developers place a high emphasis on project heroics, thinking that the certain kinds of heroics can be beneficial (Bach 1995). But I think that emphasizing heroics in any form usually does more harm than good. In the case study, mid-level management placed a higher premium on can-do attitudes than on steady and consistent progress and meaningful progress reporting. The result was a pattern of scheduling brinkmanship in which impending schedule slips weren't detected, acknowledged, or reported up the management chain until the last minute. A small development team held an entire company hostage because they wouldn't admit that they were having trouble meeting their schedule. An emphasis on heroics encourages extreme risk taking and discourages cooperation among the many stakeholders in the software-development process.

Some managers encourage this behavior when they focus too strongly on can-do attitudes. By elevating can-do attitudes above accurate-and-sometimes-gloomy status reporting, such project managers undercut their ability to take corrective action. They don't even know they need to take corrective action until the damage has been done. As Tom DeMarco says, can-do attitudes escalate minor setback into true disasters (DeMarco 1995).

#5: Adding people to a late project. This is perhaps the most classic of the classic mistakes. When a project is behind, adding people can take more productivity away from existing team members than it adds through new ones. Fred Brooks likened adding people to a late project to pouring gasoline on a fire (Brooks 1975).

#6: Noisy, crowded offices. Most developers rate their working conditions as unsatisfactory. About 60 percent report that they are neither sufficiently quiet nor sufficiently private (DeMarco and Lister 1987). Workers who occupy quiet, private offices tend to perform significantly better than workers who occupy noisy, crowded work bays or cubicles. Noisy, crowded work environments lengthen development schedules.

#7: Friction between developers and customers. Friction between developers and customers can arise in several ways. Customers may feel that developers are not cooperative when they refuse to sign up for the development schedule that the customers want, or when they fail to deliver on their promises. Developers may feel that customers unreasonably insisting on unrealistic schedules or requirements changes after requirements have been baselined. There might simply be personality conflicts between the two groups.

The primary effect of this friction is poor communication, and the secondary effects of poor communication include poorly understood requirements, poor user-interface design, and, in the worst case, customers' refusing to accept the completed product. On average, friction between customers and software developers is so severe that both parties consider canceling the project (Jones 1994). Such friction is time-consuming to overcome, and it distracts both customers and developers from the real work of the project.

#8: Unrealistic expectations. One of the most common causes of friction between developers and their customers or managers is unrealistic expectations. In Case Study 3-1, Bill had no reason to think that the Giga-Quote program could be developed in six months except for the fact that the company needed it in that amount of time. Mike's failure to correct that unrealistic expectation was a major source of problems. In other cases, project managers or developers ask for trouble by getting funding based on overly optimistic schedule estimates. Sometimes they promise a pie-in-the-sky feature set. Although unrealistic expectations do not in themselves lengthen development schedules, they contribute to the perception that development schedules are too long, and that can be almost as bad. A Standish Group survey listed realistic expectations as one of the top five factors needed to ensure the success of an in-house business-software project (Standish Group 1994).

#9: Lack of effective project sponsorship. High-level project sponsorship is necessary to support many aspects of rapid development including realistic planning, change control, and the introduction of new development practices. Without an effective project sponsor, other high-level personnel in your organization can force you to accept unrealistic deadlines or make changes that undermine your project. Australian consultant Rob Thomsett argues that lack of an effective project sponsor virtually guarantees project failure (Thomsett 1995).

#10: Lack of stakeholder buy-in. All of the major players in a software-development effort must buy in to the project. That includes the executive sponsor, team leader, team members, marketing, end-users, customers, and anyone else who has a stake in it. The close cooperation that occurs only when you have complete buy-in from all stakeholders allows for precise coordination of a rapid development effort that is impossible to attain without good buy-in.

#11: Lack of user input. The Standish Group survey found that the number one reason that IS projects succeed is because of user involvement (Standish Group 1994).

#12: Politics placed over substance. Larry Constantine reported on four teams that had four different kinds of political orientations (Constantine 1995a). "Politicians" specialized in "managing up," concentrating on relationships with their managers. "Researchers" concentrated on scouting out and gathering information. "Isolationists" kept to themselves, creating project boundaries that they kept closed to non-team members. "Generalists" did a little bit of everything: they tended their relationships with their managers, performed research and scouting activities, and coordinated with other teams through the course of their normal workflow. Constantine reported that initially the political and generalist teams were both well regarded by top management. But after a year and a half, the political team was ranked dead last. Putting politics over results is fatal to speed-oriented development.

#13: Wishful thinking. I am amazed at how many problems in software development boil down to wishful thinking. How many times have you heard statements like these:

"None of the team members really believed that they could complete the project according to the schedule they were given, but they thought that maybe if everyone worked hard, and nothing went wrong, and they got a few lucky breaks, they just might be able to pull it off."

"Our team hasn't done very much work to coordinate the interfaces among the different parts of the product, but we've all been in good communication about other things, and the interfaces are relatively simple, so it'll probably take only a day or two to shake out the bugs."

"We know that we went with the low-ball contractor on the database subsystem and it was hard to see how they were going to complete the work with the staffing levels they specified in their proposal. They didn't have as much experience as some of the other contractors, but maybe they can make up in energy what they lack in experience. They'll probably deliver on time."

"We don't need to show the final round of changes to the prototype to the customer. I'm sure we know what they want by now."

"The team is saying that it will take an extraordinary effort to meet the deadline, and they missed their first milestone by a few days, but I think they can bring this one in on time."

Wishful thinking isn't just optimism. It's closing your eyes and hoping something works when you have no reasonable basis for thinking it will. Wishful thinking at the beginning of a project leads to big blowups at the end of a project. It undermines meaningful planning and may be at the root of more software problems than all other causes combined.

Process

Process-related mistakes slow down projects because they squander people's talents and efforts. Here are some of the worst process-related mistakes.

#14: Overly optimistic schedules. The challenges faced by someone building a three-month application are quite different than the challenges faced by someone building a one-year application. Setting an overly optimistic schedule sets a project up for failure by underscoping the project, undermining effective planning, and abbreviating critical upstream development activities such as requirements analysis and design. It also puts excessive pressure on developers, which hurts developer morale and productivity. This was a major source of problems in Case Study 3-1.

#15: Insufficient risk management. Some mistakes have been made often enough to be considered classics. Others are unique to specific projects. As with the classic mistakes, if you don't actively manage risks, only one thing has to go wrong to change your project from a rapid-development project to a slow-development one. Failure to manage risks is one of the most common classic mistakes.

#16: Contractor failure. Companies sometimes contract out pieces of a project when they are too rushed to do the work in-house. But contractors frequently deliver work that's late, that's of unacceptably low quality, or that fails to meet specifications (Boehm 1989). Risks such as unstable requirements or ill-defined interfaces can be magnified

when you bring a contractor into the picture. If the contractor relationship isn't managed carefully, the use of contractors can slow a project down rather than speed it up.

#17: Insufficient planning. If you don't plan to achieve rapid development, you can't expect to achieve it.

#18: Abandonment of planning under pressure. Projects make plans and then routinely abandon them when they run into schedule trouble (Humphrey 1989). The problem isn't so much in abandoning the plan as in failing to create a substitute and then falling into code-and-fix mode instead. In Case Study 3-1, the team abandoned its plan after it missed its first delivery, and that's typical. The result was that work after that point was uncoordinated and awkward--to the point that Jill even started working on a project for her old group part of the time and no one even knew it.

#19: Wasted time during the fuzzy front end. The "fuzzy front end" is the time before the project starts, the time normally spent in the approval and budgeting process. It's not uncommon for a project to spend months or years in the fuzzy front end and then to come out of the gates with an aggressive schedule. It's much easier and cheaper and less risky to save a few weeks or months in the fuzzy front end than it is to compress a development schedule by the same amount.

#20: Shortchanged upstream activities. Projects that are in a hurry try to cut out nonessential activities, and since requirements analysis, architecture, and design don't directly produce code, they are easy targets. On one disaster project that I took over, I asked to see the design. The team lead told me, "We didn't have time to do a design."

Also known as "jumping into coding," the results of this mistake are all too predictable. In the case study, a design hack in the bar-chart report was substituted for quality design work. Before the product could be released, the hack work had to be thrown out and the higher quality work had to be done anyway. Projects that skimp on upstream activities typically have to do the same work downstream at anywhere from 10 to 100 times the cost of doing it properly in the first place (Fagan 1976; Boehm and Papaccio 1988). If you can't find the 5 extra hours to do the job right the first time, where are you going to find the 50 extra hours to do it right later?

#21: Inadequate design. A special case of shortchanging upstream activities is inadequate design. Rush projects undermine design by not allocating enough time for it and by creating a pressure-cooker environment that makes thoughtful consideration of design alternatives difficult. The design emphasis is on expediency rather than quality, so you tend to need several ultimately time-consuming design cycles before you finally complete the system.

#22: Shortchanged quality assurance. Projects that are in a hurry often cut corners by eliminating design and code reviews, eliminating test planning, and performing only perfunctory testing. In the case study, design reviews and code reviews were given short shrift in order to achieve a perceived schedule advantage. As it turned out, when the project reached its feature-complete milestone it was still too buggy to release for five more months. This result is typical. Short-cutting a day of QA activity early in the project is likely to cost you 3 to 10 days of activity downstream (Jones 1994). This inefficiency undermines development speed.

#23: Insufficient management controls. In the case study, there were few management controls in place to provide timely warnings of impending schedule slips, and the few controls there were in place at the beginning were abandoned once the project ran into trouble. Before you can keep a project on track, you have to be able to tell whether it's on track.

#24: Premature or too frequent convergence. Shortly before a product is scheduled to be released there is a push to prepare the product for release--improve the product's performance, print final documentation, incorporate final help-system hooks, polish the installation program, stub out functionality that's not going to be ready on time, and so on. On rush projects, there is a tendency to force convergence early. Since it's not possible to force the product to converge when desired, some rapid development projects try to force convergence a half dozen times or more before they finally succeed. The extra convergence attempts don't benefit the product. They just waste time and prolong the schedule.

#25: Omitting necessary tasks from estimates. If people don't keep careful records of previous projects, they forget about the less visible tasks, but those tasks add up. Omitted effort often adds about 20 to 30 percent to a development schedule (van Genuchten 1991).

#26: Planning to catch up later. If you're working on a six-month project, and it takes you three months to meet your two-month milestone, what do you do? Many projects simply plan to catch up later, but they never do. You learn more about the product as you build it, including more about what it will take to build it. That learning needs to be reflected in the schedule.

Another kind of reestimation mistake arises from product changes. If the product you're building changes, the amount of time you need to build it changes too. In Case Study 3-1, major requirements changed between the original proposal and the project start without any corresponding reestimation of schedule or resources. Piling on new features without adjusting the schedule guarantees that you will miss your deadline.

#27: Code-like-hell programming. Some organizations think that fast, loose, all-as-you-go coding is a route to rapid development. If the developers are sufficiently motivated, they reason, they can overcome any obstacles. For reasons that will become clear throughout this book, this is far from the truth. The entrepreneurial model is often a cover for the old code-and-fix paradigm combined with an ambitious schedule, and that combination almost never works. It's an example of two wrongs not making a right.

Product

Here are some classic mistakes are related to the way the product is defined.

#28: Requirements gold-plating. Some projects have more requirements than they need right from the beginning. Performance is stated as a requirement more often than it needs to be, and that can unnecessarily lengthen a software schedule. Users tend to be less interested in complex features than marketing and development are, and complex features add disproportionately to a development schedule.

#29: Feature creep. Even if you're successful at avoiding requirements gold-plating, the average project experiences about a 25-percent change in requirement over its lifetime (Jones 1994). Such a change can produce at least a 25-percent addition to the software schedule, which can be fatal to a rapid development project.

#30: Developer gold-plating. Developers are fascinated by new technology and are sometimes anxious to try out new features of their language or environment or to create their own implementation of a slick feature they saw in another product--whether or not it's required in their product. The effort required to design, implement, test, document, and support features that are not required lengthens the schedule.

#31: Push me, pull me negotiation. One bizarre negotiating ploy occurs when a manager approves a schedule slip on a project that's progressing slower than expected and then adds completely new tasks after the schedule change. The underlying reason for this is hard to fathom because the manager who approves the schedule slip is implicitly acknowledging that the schedule was in error. But once the schedule has been corrected, the same person takes explicit action to make it wrong again. This can't help but undermine the schedule.

#32: Research-oriented development. Seymour Cray, the designer of the Cray supercomputers, says that he does not attempt to exceed engineering limits in more than two areas at a time because the risk of failure is too high (Gilb 1988). Many software projects could learn a lesson from Cray. If your project strains the limits of computer science by requiring the creation of new algorithms or new computing practices, you're not doing software development; you're doing software research. Software-development schedules are reasonably predictable; software research schedules are not even theoretically predictable.

If you have product goals that push the state of the art--algorithms, speed, memory usage, and so on--you should expect great uncertainty in your scheduling. If you're pushing the state of the art and you have any other weaknesses in your project--personnel shortages, personnel weaknesses, vague requirements, unstable interfaces with outside contractors--you can throw predictable scheduling out the window. If you want to advance the state of the art, by all means, do it. But don't expect to do it rapidly!

Technology

The remaining classic mistakes have to do with the use and misuse of modern technology.

#33: Silver-bullet syndrome. In the case study there was too much reliance on the advertised benefits of previously unused technologies (report generator, object oriented design, and C++) and too little information about how well they would do in this particular development environment. When project teams latch onto a single new methodology or new technology and expect it to solve their schedule problems, they are inevitably disappointed (Jones 1994).

#34: Overestimated savings from new tools or methods. Organizations seldom improve their productivity in giant leaps, no matter how good or how many new tools or methods they adopt. Benefits of new practices are partially offset by the learning curves

associated with them, and learning to use new practices to their maximum advantage takes time. New practices also entail new risks, which you're likely to discover only by using them. You are more likely to experience slow, steady improvement on the order of a few percent per project than you are to experience dramatic gains. The team in Case Study 3-1 should have planned on, at most, a 10-percent gain in productivity from the use of the new technologies instead of assuming that they would nearly double their productivity.

A special case of overestimated savings arises when projects reuse code from previous projects. This can be a very effective approach, but the time savings is rarely as dramatic as expected.

#35: Switching tools in the middle of a project. This is an old standby that hardly ever works. Sometimes it can make sense to upgrade incrementally within the same product line, from version 3 to version 3.1 or sometimes even to version 4. But the learning curve, rework, and inevitable mistakes made with a totally new tool usually cancel out any benefit when you're in the middle of a project.

#36: Lack of automated source-code control. Failure to use automated source-code control exposes projects to needless risks. Without it, if two developers are working on the same part of the program, they have to coordinate their work manually. They might agree to put the latest versions of each file into a master directory and to check with each other before copying files into that directory. But someone always overwrites someone else's work. People develop new code to out-of-date interfaces and then have to redesign their code when they discover that they were using the wrong version of the interface. Users report defects that you can't reproduce because you have no way to recreate the build they were using. On average, source code changes at a rate of about 10 percent per month, and manual source-code control can't keep up (Jones 1994).

Table 3-1 contains a complete list of classic mistakes.

Table 3-1. Summary of Classic Mistakes

People-Related Mistakes	Process-Related Mistakes	Product-Related Mistakes	Technology-Related Mistakes
1. Undermined motivation	14. Overly optimistic schedules	28. Requirements gold-plating	33. Silver-bullet syndrome
2. Weak personnel	16. Insufficient risk management	29. Feature creep	34. Overestimated savings from new tools or methods
3. Uncontrolled problem employees	17. Contractor failure Insufficient planning	30. Developer gold-plating	35. Switching tools in the middle of a project
4. Heroics	18. Abandonment of planning under pressure	31. Push me, pull me negotiation	36. Lack of automated source-code control
5. Adding people to a late project		32. Research-oriented development	
6. Noisy, crowded			

offices 7. Friction between developers and customers 8. Unrealistic expectations 9. Lack of effective project sponsorship 10. Lack of stakeholder buy-in 11. Lack of user input 12. Politics placed over substance 13. Wishful thinking	19. Wasted time during the fuzzy front end 20. Shortchanged upstream activities 21. Inadequate design 22. Shortchanged quality assurance 23. Insufficient management controls 24. Premature or too frequent convergence 25. Omitting necessary tasks from estimates 26. Planning to catch up later 27. Code-like-hell programming		