Classic Mistakes

I first came across Steve McConnell's Rapid Development when I found it on a colleague's desk last year. Inside were several pseudo-bookmarks where people have left off reading and had probably forgotten to return. If I remember correctly, I myself have stuck in a calling card and some restaurant's flyer. Anyway, my fave part of the book has got to be the bits on Classic Mistakes. They're classic so, in one way or another, I had personally come across them in my own projects. As I went through the list, I could recall going "ooh" nearly cringing in recognition. But the reason for having this list is not to rub salt to the wound. Rather, this list highlights what you need to consider during project planning and risk management. It includes 36 classic mistakes categorized into 4 areas — People, Process, Product and Technology.

People

1. **undermined motivation** – study after study has shown that motivation probably has a larger effect on productivity and quality than any other factor. some examples of stuff that undermines motivation include hokey pep talks, requiring overtime, management going on long vacations while team worked through holidays, providing end-of-project bonuses that didn't compensate the overtime effort rendered.
2. **weak personnel** – hiring from the bottom of the barrel can get project off to a quick start but doesn't set it up for rapid completion.
3. **uncontrolled problem employees** – 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). seeing nothing being done about problematic team mates and suffering for their lapses can lower morale.
4. **heroics** – emphasizing heroics in any form usually does more harm than good. an emphasis on heroics encourages extreme risk taking and discourages cooperation among the many stakeholders in the software development process. by elevating can-do attitudes above accurate-and-sometimes-gloomy status reporting, you miss out on taking the appropriate corrective action.
5. **adding people to a late project** – most classic. think brooks's law!
6. **noisy, crowded office** – workers who occupy quiet, private offices tend to perform significantly better than workers who occupy noisy, crowded work bays or cubicles. a related link I tweeted before – http://tinyurl.com/7qyvta
7. **friction between developers and customers** – customers may feel that devs are not cooperative; devs may feel that customers are unreasonable. effect of this friction is poor communication which leads to poorly understood requirements, poor ui design, and in worst case, customers' refusal to accept the product.
8. **unrealistic expectations** – inability to correct unrealistic expectation, overly optimistic schedule estimates. by themselves they do not lengthen the devt schedule. but there's just trouble when unrealistic expectations in schedule or in features are compared with reality… it gives off the perception that it has gone too long or the features too insufficient.

9. **lack of effective project sponsorship** – without an effective executive sponsor, other high-level personnel in your org can force you to accept unrealistic deadlines or make changes that undermine your project.
10. **lack of stakeholder buy-in** – all major players in a sw devt effort must buy in to the project — executive sponsor, team leader, team members, marketing staff, end-users, customers. good buy-in leads to close cooperation and precise coordination of the rapid devt effort.
11. **lack of user input** – projects without early end-user involvement risk misunderstanding the project's requirements and are vulnerable to time-consuming feature creep later in the project.
12. **politics placed over substance**
13. **wishful thinking** – this isn't just optimism. it's closing your eyes and hoping something works when you have no reasonable basis for thinking it will. It undermines meaningful planning. causes trouble when reality blows up in your face.

Process

1. **overly optimistic schedules** – 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. puts excessive pressure on the team, which hurts long-term team morale and productivity.
2. **insufficient risk management** – only one thing has to go wrong to change your project from a rapid-development project to a slow-development one.
3. **contractor failure** – getting contractors is sometimes done to alleviate pressure off the team. but contractors frequently deliver work that's late, 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.
4. **insufficient planning** – if you don't plan to achieve rapid devt, you can't expect to achieve it. it's wishful thinking to think it will magically happen on its own.
5. **abandonment of planning under pressure** – project teams 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. after abandoning the plan, the work becomes uncoordinated and awkward. and it's downright confusing when you've no idea on what's happening and on who's doing what.
6. **wasted time during the fuzzy front end** – it's not uncommon for a project to spend months or years in the fuzzy front end and then 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 devt schedule by the same amount.
7. **shortchanged upstream activities** – the results of this mistake — also known as "jumping into coding" — are all too predictable. rework! 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 (Pagan 1976; Boehm and Papaccio 1988)

8. **inadequate design** – special case of the former; rush projects undermine design by not allocating enough time for it and creating a pressure cooker envt that makes thoughtful consideration of design alternatives difficult. the design emphasis is on expediency rather than quality. you end up with a design product which doesn't actually help that much during implementation.
9. **shortchanged quality assurance** – cut corners by eliminating design and code reviews, eliminating test planning and performing only perfunctory testing. result: product still too buggy to release. Shortcutting 1 day of QA activity early in the project is likely to cost you from 3 to 10 days of activity downstream (Jones 1994).
10. **insufficient management controls** – few mgt controls to provide timely warnings of impending schedule slips; before you can keep a project on track, you have to be able to tell whether it's on track in the first place.
11. **premature or overly frequent convergence** – forcing convergence too early results to more effort as separate parts of the system aren't ready and the people themselves aren't prepared. extra convergence attempts don't benefit the product; only prolongs the schedule.
12. **omitting necessary tasks from estimates** – you still have to do these anyway. but without these considered into your estimates, you'll look delayed or over budget.
13. **planning to catch up later** – most likely, there is no later.
14. **code-like-hell programming** – it is really just 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

1. **requirements gold-plating** – users tend to be less interested in complex features than marketing and development are, and complex features add disproportionately to a development schedule.
2. **feature creep** – even if you're successful at avoiding requirement gold-plating, the avg project experiences about a 25% change in requirements over its lifetime (Jones 1994). there's no escape from changing requirements.
3. **developer gold-plating** – devs are fascinated by new technology and are sometimes anxious to try out new features of their language or envt or to create their own implementation of a slick feature they saw in another product — whether or not it's required in their product.
4. **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. it's like rubbing salt to a wound.
5. **research-oriented development** – 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.

Technology

1. **silver-bullet syndrome** – too much reliance on the advertised benefits of previously unused technologies and too little information about how well they would do in this particular development environment. when project teams latch onto a single new practice, new technology, or rigid process and expect it to solve their schedule problems, they are inevitably disappointed (Jones 1994).
2. **overestimated savings from new tools or methods** – orgs seldom improve their productivity in giant leaps, no matter how many new tools or methods they adopt or how good they are. 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.
3. **switching tools in the middle of a project** – learning curve, rework and inevitable mistakes made with a new tool usually cancel out any benefit when you're in the middle of a project.
4. **lack of automated source-code control** – exposes the project to needless risks e.g., problems with versioning, one dev overwriting someone's updates, defects can't easily be replicated since there's 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).

[KC: These notes were previously blogged back in 2009. The related posts can be found with the tag "classic mistakes", and here's a link: http://testkeis.wordpress.com/?s=classic+mistakes.]