Classic Mistakes

Some of the “big” classic mistakes in the case study ...

1. Undermined motivation

- hokey pep talk at beginning of project
- requiring overtime in middle
- going on vacation while team worked through holidays
- end of project bonuses came out to < $1 / overtime hr
Classic Mistakes

Some of the “big” classic mistakes in the case study …

2. Weak personnel

- hiring based on who could get started fastest
- two people recommended against Chip
- he ended up causing friction

3. Wishful thinking

- hoping it works out without any basis to think so

  e.g.: “None of the team members really believed that they could complete the project according to the schedule they were given, but they thought 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”
4. Uncontrolled problem employees

- team knew Chip was a problem, Mike did nothing
- ended up having to redo all of Chip’s work

5. Overestimated savings from new tools / methods

- team thought C++ and report tool would save 3 months of work
- this would be about 25% savings!
**Classic Mistakes**

_Some of the “big” classic mistakes in the case study …_

6. **Shortchanging upstream activities**
   - aggressive schedules often cut out “non-essentials” (non-coding activities like requirements, design, etc.)
   - skimping on these often requires same work later at **10 to 100** times the cost (re-work)

---

7. **Abandonment of planning under pressure**
   - after team missed first delivery, work became uncontrolled and awkward
   - Jill even started to work on a project for her old team and no one knew it
Classic Mistakes

Some of the “big” classic mistakes in the case study …

8. Heroics

- asking for “herculean” efforts instead of steady progress
- Bill kept asking for more effort (implying team wasn’t working hard enough!)

Classic Mistakes

Other classic mistakes

- Lack of project sponsorship / buy-in
- Lack of user input
- Feature creep … (25% change on avg over project lifetime)
- Gold plating … (not stopping when “good enough”)
- Push-me, Pull-me … (schedule slip but add more tasks)
- Lack of appropriate development tools
- Overly optimistic schedules
- Premature / too-frequent convergence
“Best” Practices / Principles

Design & Development “Principles”

Software Engineering methodologies come and go

- E.g., Waterfall, Spiral, RUP, RAD, XP, Kanban, Lean, Agile, Scrum, ...

*But many of the underlying “principles” & “best practices” seem to stick around*
Examples: **Software Engineering (generally)**

1. Figure out what you need to (do before you do it)
   - Requirements (disagreements on detail needed)
   - Mock-ups, Prototyping
   - Channel the user (or even be the user … aka “dogfooding”)

2. Visibility is good
   - keep customer involved, get feedback, iterate/prototype/refine

3. Be organized and structured, but adapt
   - Have a plan, follow it, change it as needed
   - But avoid myth-making

Examples: **Software Engineering (generally)**

4. Keep track of where you are and where you’re going
   - e.g., milestones, roadmaps, wbs, ...
   - ... and when you’ll get there (estimation)

5. Communicate
   - e.g., stand-up meetings, planning/review/retrospective meetings, documentation, etc.

6. Validate (with actual users)
   - get your product in front of real users early and often
   - Incorporate this into your process
Examples: *Software Development* (generally)

1. Use useful tools
   - Versioning, building, integrating, testing, tracking issues, ...

2. Test and review your software
   - Unit, integration, system, usability, stress, etc.

3. Consider maintenance
   - Part of “*What you produce, others will consume*”
   - Assume someone else will need to understand your work

---

Examples: *Software Development* (generally)

4. Focus on value
   - Decisions should be made based on providing value to users
   - Focus on the reason you are building the software

5. Do what is needed (“*You aren’t going to need it*”)
   - Don’t implement things until you actually need them
   - You probably won’t need it, or it will be different when you do
   - Won’t save time now by doing something you don’t need now
**Examples: Software Design**

1. Keep it simple, stupid!
   - As simple as possible, no simpler
   - Simpler is easier to understand, maintain, extend
   - Simpler is harder than “quick and dirty”
   - Simpler is harder than “elegant” (i.e., complex)

2. Have an (intentional) design
   - No design is not the opposite of over-engineered
   - Let the design evolve / improve it as needed (refactoring)
   - Avoid “gold-plating”

**Examples: Software Design**

3. Be a defensive programmer
   - Fail fast principle
   - Consider and mitigate what can go wrong (e.g., check inputs)
   - Opposite: Design by contract

4. Divide and conquer
   - Modularity (decomposition into smaller “modules”)
   - Avoid duplication (DRY)
   - Avoid unnecessary dependencies (loose coupling)
   - Use standard/vetted approaches (e.g., design patterns)

And lots more …