CS 428
FACTS & FALLACIES OF SOFTWARE ENGINEERING (CHAPTER 1)

Winter 2022 – Bruce F. Webster

CHAPTER 1: MANAGEMENT
SECTION 1: PEOPLE

- #1: The most important factor in software work is not the tools and techniques used by the programmers, but rather the quality of the programmers themselves.
  - An inadequate process/tech base with great people will succeed in situations where a great process/tech base with inadequate people will not. (Webster)
  - Why is this ignored/neglected so much? Because it is hard to find, screen for, and hire great people, whereas it is easy to decide to use new technologies and new methodologies.

CHAPTER 1, SECTION 1: PEOPLE

- #2: The best programmers are up to 28 times better than the worst programmers, according to “individual differences” research. Given that their pay is never commensurate, they are the biggest bargains in the software field.
  - Studies show anywhere from 5:1 [too low!] to 28:1 differences, even controlling for education and experience.
  - Goal, then, should be to find, hire, and keep the best programmers. Instead, we often drive them away (the “Dead Sea Effect”).
  - The challenge is identifying them – their talent often only becomes apparent in real-world settings.

CHAPTER 1, SECTION 1: PEOPLE

- #3: Adding people to a late project makes it later.
  - Brooks' Law, of course.
  - As per Brooks, there are in fact circumstances when adding people can help: important subject-matter expertise, replacement after departure of key personnel, etc.
  - But the management impulse to try to get a late project back on track by adding more people to it almost always backfires.

CHAPTER 1, SECTION 1: PEOPLE

- #4: The working environment has a profound impact on productivity and product quality.
  - Once you've hired great people, you want to put them in "an environment…that facilitates thinking." – cites *Peopleware*
  - Everyone acknowledges this, but management tends to focus on what's economic and easy rather than what will help developers to be most productive.

CHAPTER 1, SECTION 2: TOOLS AND TECHNIQUES

- #5: Hype is the plague on the house of software. Most software tool and technique improvements account for about a 5 to 35 percent increase in productivity and quality. But at one time or another, most of these same improvements have been claimed by someone to have "order of magnitude" [10x – not 100% but 1000%] benefits.
  - This is, of course, the "silver bullet" or "Laetrile" syndrome, and it crops up constantly.
  - Be wary of any claims for tools or methodologies that will "revolutionize" programming (or even "replace" it).

CHAPTER 1, SECTION 2: TOOLS AND TECHNIQUES

- #6: Learning a new tool or technique actually *lowers* programmer productivity and product quality initially. The eventual benefit is achieved only after this learning curve is overcome. Therefore, it is worth adopting new tools and techniques, but only (a) if their value is seen realistically and (b) if patience is used in measuring benefits.
  - This is so important to understand and is so often ignored.
  - The great pitfall: adopting a new tool or technique for a mission-critical project.
  - Even worse: changing to a new tool or technique in the middle of a critical project.
  - Worse yet: making critical business decisions based on unrealistic expectations for that new tool or technique.
  - "Start out stupid and work up from there." – Bruce Henderson

CHAPTER 1, SECTION 2: TOOLS AND TECHNIQUES

- #7: Software developers talk a lot about tools. They evaluate quite a few, buy a fair numbers, and use practically none.
  - How many development/language environments do you have downloaded on your computer(s)? How many are you using to actually write meaningful software?
  - How many software tools – outside of IDEs – do you have and are you making real-world use of? Why or why not?
  - Common issue: unrealistic deadlines make us wary of trying out tools that we aren't familiar with – we'd rather stick to the tried and true.
  - What have you learned about certain tools just through your projects in this class?

CHAPTER 1, SECTION 3: ESTIMATION

- #8: One of the two most common causes of runaway projects is poor estimation. [Spoiler: the other is unstable requirements, #23]
    - Runaway project: the pattern we've discussed (gets close to deadline, slips, gets close to new deadline, slips again, rinse and repeat)
    - Back to the core issue: while we may see the end goal, we usually lack the processes and experience to know how long it will take to get there (cf. Armour)
    - "We don't know what constitutes a good estimation approach, one that can yield decent estimates with good confidence that they will really predict when a project will be completed and how much it will cost."
    - "[Certain] projects became runaways because the estimation targets to which they were being managed were largely unreal to begin with."

CHAPTER 1, SECTION 3: ESTIMATION

- #9: Most software estimates are performed at the beginning of the life cycle. This makes sense until we realize that estimates are obtained before the requirements are defined and thus before the problem is understood. Estimation, therefore, usually occurs at the wrong time.
  - How can you estimate solution time and cost if you don't yet know what problem you are going to be solving?
  - This is Armour writ large: the value of a new software system is (usually) directly proportional to the amount of exploration and invention it will require (its novelty), which in turn increases the amount of non-productive exploration that will be required to find a solution.

CHAPTER 1, SECTION 3: ESTIMATION

- #10: Most software estimates are made either by upper management or by marketing, not by the people who will build the software or their managers. Estimation is, therefore, done by the wrong people.
  - This is true far more often than it should be. It's like management/marketing deciding how long it should take to build a dam or a skyscraper with little input from architects, civil engineers, and construction managers.
  - Compounding the problem are the factors that Brooks talks about: the natural optimism of engineers, and the unwillingness of managers to give bad (i.e., honest) news.
  - On the other hand, when you do have honesty (and incredulity) coming up from the trenches, it often gets blocked by the 'thermocline of truth'.

CHAPTER 1, SECTION 3: ESTIMATION

- #11: Software estimates are rarely adjusted as the project proceeds. Thus, those estimates done at the wrong time by the wrong people are usually not corrected.
    - At least, they're not corrected until it because blindingly obvious to even top management that the original deadline will not be met.
    - As per Brooks (and me), in corporate/organizational projects, that seems to happen usually around 3 weeks before delivery – and that says a lot about the 'magic thinking' and outright denial that goes on in mid- to upper management, as well as the absence of any useful project completion metrics.
    - Commercial software firms (including startups) often have a more realistic sense of where things stand – but not always.
    - Always remember Brooks: "Take no small slips."

CHAPTER 1, SECTION 3: ESTIMATION

- #12: Since estimates are so faulty, there is little reason to be concerned when software projects do not meet estimated targets. But everyone is concerned anyway.
  - We manage by schedule rather than by more relevant/useful metrics:
    - Product functionality (feature availability)
    - Issue (find/fix ratio)
    - Risk (find/fix ratio)
    - Business objectives (business process improvement to date)
    - Quality (quality attributes achieved to date)
  - People are trying so hard to achieve (impossible) schedules that they are willing to sacrifice completeness (functionality) and quality (reliability/performance) to get there.
  - Note: DevOps appears to be an attempt to sidestep the entire 'schedule' focus by instead aiming for constant development and deployment.

CHAPTER 1, SECTION 3: ESTIMATION

- #13: There is [often] a disconnect between management and their programmers.
  - In one research study of a project that failed to meet its estimate and was seen by its management as a failure (schedule, budget), the technical participants saw it as the most successful project they had ever worked on (great design, great team, tough problems solved, resulting functionality, lack of defects).
  - When asked why project was late, programmers said: unrealistic estimates, lack of expert resources, poorly understood scope, late start, *all at the start of the project*. ("The important mistakes are made the first day." - Spinrad)
  - Study: "projects where no estimates were prepared at all fared best on productivity." (Jeffery & Lawrence, 1985)

CHAPTER 1, SECTION 3: ESTIMATION

- #14: The answer to a feasibility study is almost always "yes".
  - "We seem to possess all-too-often incurable optimism. It's as if, since no one has ever been able to solve the problems we are able to solve, we believe that no new problem is too tough for us to solve. And, astonishingly often, that is true. But there are times when it is not, times when that optimism gets us into a world of trouble."
  - Echoing Brooks on project estimation: "We believe we will instantly produce software without errors, and then find that the error-removal phase often takes more time than system analysis, design, and coding put together."
  - "There is such a time gap between getting the wrong answer to a feasibility study and the discovery that it really was the wrong answer, that we rarely connect those two events."
  - Jerry Weinberg at software engineering conference (1500 attendees): "How many of you have participated in a feasibility study where the answer came back 'No'?" No one raised their hand.

CHAPTER 1, SECTION 4: REUSE

- #15: Reuse-in-the-small (libraries of subroutines) began nearly [65] years ago and is a well-solved problem.
- #16: Reuse-in-the-large (components) remains a mostly unsolved problem, even though everyone agrees it is important and desirable.
- #17: Reuse-in-the-large works best in families of related systems and thus is domain-dependent. This narrows the potential applicability of reuse-in-the-large.
  - Most successful reuse-in-the-large found in things like graphics and physics engines.
  - Reuse parallels buy-vs-build decision: buy/license/reuse components that get you onto equal footing with your competitors, invent/develop that which makes you better.
  - Even among "related systems" in the same domain, different systems require different abstractions (cf. universal "loan" object at Fannie Mae)

CHAPTER 1, SECTION 4: REUSE

- #18: There are two "rules of three" in reuse: (a) it is three times as difficult to build reusable components as single use components, and (b) a reusable component should be tried it in three different applications before it will be sufficiently general to accept into a reuse library.
- #19: Modification of reused code is particularly error-prone. [Q: Why?] If more than 20-25% of a component is to be revised, rewrite it from scratch.
  - It is almost always a mistake to modify packaged, vendor-produced software.
- #20: Design pattern reuse is one solution to the problems inherent in code reuse.
  - Design patterns emerge from practice, not from theory.

CHAPTER 1, SECTION 5: COMPLEXITY

- #21: For every 25% increase in problem complexity, there is a 100% increase in complexity of the software solution.
  - This explains so many of the other facts in this book, including both poor estimation by management and excessive optimism by engineers.
  - We ignore this at our peril.
- #22: 80% of software work is intellectual. A fair amount of it is creative. Very little of it is clerical.
  - Which is why software engineering has never become automated (and AI won't replace us, either).
  - Once again, back to Armour: exploration and knowledge-encoding, with a solution that has to achieve requisite functionality, performance, and reliability.

WHAT'S AHEAD

- By midnight on Saturday (03/05)
  - Team: status report #6, test plan
  - Individual: podcast #4
- By next Monday (03/07)
  - Read *Facts and Fallacies* chapter 2
  - Keep working on Webster #6 (we'll cover the first set in class next week)
- By two weeks from this Saturday (03/19)
  - Code reviews (T/F "exam")
- Three weeks from today (03/21): Work-in-progress demos, midterm prep
- Four weeks from today (03/28): Midterm
- Six weeks from today (04/11, last day of class): Final demos