CSE 403
Lecture 25

Scheduling and Planning a Large Project

Reading:

_The Mythical Man-Month_, Ch. 2, by F. Brooks

slides created by Marty Stepp
http://www.cs.washington.edu/403/
Revisited: Software is hard

• Historically, ~ 85% of software projects "fail." Why?
  – management sets unrealistic expectations; devs don't correct them
  – overestimating the positive impact of shiny new tools and hardware
  – hired developers based on availability despite warning signs
  – personality conflicts between developers
  – changes in rate structure requirements in middle of work
  – one delay causes another (dev delay leads to test delay, etc.)
  – hacks and shortcuts
  – developers end up working "death marches" (6-day, 10-hour weeks)
  – overestimating how nearly done you are ("I'm 90% there!")
  – software written doesn't match the spec
  – developer time taken away by other tasks
  – tons of bugs come out in testing
  – developers don't listen to testers; ignore severity of bugs reported
  – management breaking promises (bonuses, time off, etc.)
Why do projects fail?

- **Fred Brooks**: Turing Award-winning Harvard professor; expert on software engineering.
  - managed development of IBM System/360
  - author of *The Mythical Man-Month*

- Brooks: "More programming projects have gone awry for lack of calendar time than for all other causes combined."

- But why do projects finish late?
  - How can we foresee/predict this happening?
  - What (if anything) can we do about it?
A late software project

- In the graphs, the project was supposed to reach milestone A in 1 month (left), but in fact it took 2 months (right).
  - How should this delay be interpreted?
  - What are the options facing the project's manager?
  - Should the manager add extra people to the development team to make up for the delay? If so, how many and why?
Interpretation #1

• Only Part A was misestimated.
  So the overall project will be 1 month late. (at right)

  – If the assumption is valid:
    • 9 man-months of work remain
    • / 2 actual months remain in which to do it
    • = 4.5 people will need to work each month
    • so add 2 workers to the existing 3.
• *The whole project estimate was low.* So the project will take twice as long as expected. (at right)

– If the assumption is valid:
  • 18 man-months of work remain
  • \( / \) 2 actual months remain in which to do it
  • \( = \) 9 people will need to work each month

• so add 6 workers to the existing 3.
The Mythical Man-Month

• If we assume the interpretation #1 was correct:
  – We must account for delays in training the new workers
  – We must partition the job into 5 pieces, to be integrated later

What is "Brooks' Law"?
  – "Adding manpower to a late software project makes it later."

-- The Mythical Man-Month
• Men/women and months are not interchangeable! When you add workers, the following costs occur:
  – must repartition the work
  – must train the new workers
  – must increase intercommunication

• What is Brooks' suggested schedule?
  – 1/3 for design
  – 1/6 for coding
  – 1/4 for unit/component testing
  – 1/4 for system testing
• Pro developers often write **50-100 lines of code** per day.
  – How can it be so low?
  – Does it change based on the programming language used?

• Factors to consider:
  – Should say, 100 lines of **correct** code per day.
    • Are we counting comments? Blank lines? Modified old lines?
  – The code must be...
    • designed
    • tested
    • code reviewed
    • checked in
    • maintained / updated
Productivity

• Factors that eat up developer time:
  – learning new systems, languages, and code
  – documentation
  – testing
  – debugging (getting stuck!)
  – meetings
  – interpersonal communication
  – code reviews
  – design reviews
  – illness
  – real life (family, pets, flat tire, etc.)
  – distraction (Facebook, etc.)
Measuring productivity

• Ways LoC can be useful:
  – when measured in the same language
  – with the same developer
  – over a long period of time

• Variations
  – Include comments / blank lines in LoC?

• Other ways to measure productivity, besides LoC:
  – LoC per month
  – "function points"
  – "eLoC" - substantive lines
  – check-ins
Why is estimating hard?

Why are we so bad at estimating how long a project will take?

• Programmers are optimists: "All will go well."
  – Programming = creative; building with "thought-stuff"
  – therefore, we do not usually imagine that things will go wrong

• We lack practice at measuring how long tasks will take
  – lose track of time while coding; forget how long it took
  – tend to focus on the time needed to finish "rough" untested code

• We fail to account ahead of time for:
  – bugs; sticking points (sometimes NO progress will be made)
  – design / redesign / refactoring
  – testing and debugging (both our code and others')
Some estimating tips

• Guess how long you think you'll actually need...
  – Then **double** (or **triple**) it.
  – Use a coarse granularity; days/weeks, not hours.

• Add time to your estimate if:
  – It involves learning any new technologies or systems.
  – It involves collaborating with others.
  – It is user-facing and therefore needs to be very robust/secure.
  – It is concurrent, network-enabled, or long-running.
  – It involves "messy" data or combining data from multiple sources.
• Looking back on your initial estimated project schedules:
  – How accurate were your initial ideas?
  – In what way are they the most "off" from what you have actually spent your time doing?
  – Do you know something now that would help you to more effectively schedule a large project in the future? If so, what?
Code maintenance

- **maintenance**: Modification or repair of a software product after it has been delivered.
  - fix bugs, performance, improve design, add features

- Maintenance is how developers spend much of their time.

- It's harder to maintain code than write your own new code.
  - "house of cards" phenomenon (don't touch it!)
  - must understand code written by another developer, or code you wrote at a different time with a different mindset
  - most developers dislike code maintenance...
Performing maintenance

• Maintenance comprises all phases of the software lifecycle.
  – gather requirements
  – design
  – implement (code)
  – test/debug
  – integrate
  – ...

• New versions of your software are subject to all constraints that were placed on the old version, and possibly more.
  – backwards compatibility is often expected / required
• It is often done as an afterthought.
  – Not enough time allocated in schedule
  – You must think ahead of time how you (or someone) will maintain code later

• Maintenance is often given to junior developers.
  – "This way they'll learn the guts of the system better."
  – Senior developers don't want to work on maintenance.
  – But junior devs don't know the system or how to maintain it.

• Result: brittle code with little conceptual or design integrity; even more maintenance headaches to come.