Software Project Challenge

Thanks to Nick Parlante for much of this handout

This handout begins our discussion of software project management, looking at famous problems and the goal of parallel development.

Software Project Traps

First we look at the basic facts and problems of software projects.

A Note of Optimism

- I will spend time warning you about the various danger areas that can wreck a software project so you can be aware of them. However, Java is a great language, OOP design works very well, and teams are well able to use their judgment to course-correct as they go. Many CS108 projects do their planning diligently, code in parallel, and then find, to their relief, that it all fits together and runs nicely -- escaping the many catastrophes described here. :-)

The Bad News -- Harder Than It Looks

- Uncertain
- Expensive
- Intangible -- hard to judge the current state
- Huge time-to-market issues
- Fad processes, gurus -- the "silver bullet" wish
- Many software projects fail. Shipping working, useful software is harder than it looks.
  - One typical "death state" looks like a big system where the parts do not fit together -- seemingly never quite debugged
  - Another big cause of problems is that the system does not solve the right problem or is otherwise not usable by its target audience

Keep Some Humbleness

- Perhaps because software looks so weightless and easy ... it leads projects to their doom on a path of complacency.
- Given the record of software catastrophes, we'll try to keep some humbleness
- Simplicity -- prefer simple, workable solutions first.
- Evolutionary -- don't try do it all in one "big bang" move. Build something that works and evolve it.
- The above ideas fit into the whole "Agile" design movement, which we'll discuss in more detail later on.

Tools That Help Us

- Languages -- with modularity and memory support to blunt some of the problems that destroy software projects
- Libraries of off the shelf code
- Tools -- Eclipse, CVS
- Techniques -- Modularity, OOP, design and testing disciplines.

Time Problems

Underestimating Time to Finish

- Everything takes longer than you think, i.e. the "Iceberg" principle
- Old joke: take the engineer estimate, double it, go up to the next unit of measure. So "2 weeks" means "4 months"
- External/Abstraction View
  - The way you think about something from the outside -- how it interfaces with other things.
  - The abstraction it presents.
- Internal/Implementation View
  - The external view + all the details and quirks of how it is actually implemented.
  - The internal view is much more complex than the external view.
- Iceberg Analogy
  - 20% of the iceberg sticks above the surface, but most of the iceberg is below the waterline.
  - With most systems, we observe the obvious part sticking above the water line and use that view in our mental models for planning. Most of the complexity is not visible.
- Optimistic Engineering
  - Maybe people attracted to engineering have a sort of optimistic, "here's how we're going to do it" viewpoint, but that tricks them in to underestimating the problems.
  - I have a lot of respect for the optimistic view to try to take make things work. It's a tempting pose to sit around and say how things are not going to work, but those people never get anything done. People who take the risk to try to do something are the only ones who actually get anything done.
- Tend To Plan in terms of External View
  - When thinking about a system, we tend to use the external view for the various components. (I believe this is an inherent feature of using a brain which must use conceptual simplifications to think about a complex world.)
- Underestimate
  - As a result, estimates invariably understate the total complexity. This happens every time.
- Allow Extra Time
  - There will always be unexpected details and problems -- allow extra time for them. This applies to any project (a paper, a widget, a movie, a vacation). There's always more work in the details than is apparent from the external view of the finished product.
  - When making schedules, allow an extra 50% to deal with the inevitable "unexpected" problems. "Under promise, over deliver"
- e.g. Making List of Tasks
  - Suppose you were to try to list out ahead of time all the tasks in a software project.
  - 1. Make a list ahead of time of all the methods and issues which a piece of software will need to deal with
  - 2. Do you really think you've thought of everything in (1)? All the unexpected bugs? All the blind alleys? It's guaranteed that there will be issues that are only revealed during the implementation. Could add a blanket "things we haven't thought of in this list" item to the list.

Exactly Late

- One trap is to procrastinate or work on non-critical parts until about N hours before the deadline, and then work really hard. The problem is that if the N estimate is a little low, the project just misses the deadline in a maddening sort of "if we only had 8 more hours" sort of way.

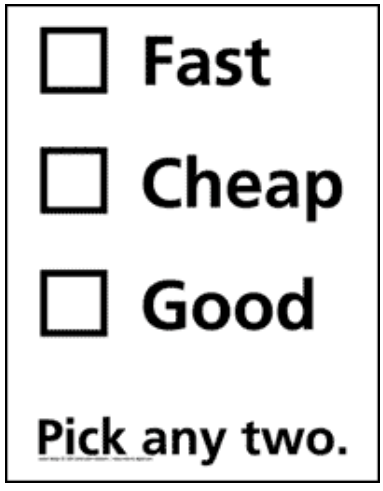
When Can you have X done?

- When planning a project, the answer to the question "when can you have X done?" is not "what is the earliest date by which you just might be able to have X done?" It is "what is a reasonable date, allowing for the usual unexpected problems, by which X can reliably be finished?" People tend to give optimistic estimates -- partly because of the iceberg principle, and partly from an honest wish to appear on the ball.

Tradeoff Problems

The Three Variables -- "Tradeoff"

- Features - Time - Quality
  - As time runs out, either features or quality will have to give.
  - Features also known as "scope"
  - Quality is the easiest to let slip, as it is hard to measure
- Classic Error: Denial
  - Denial. The project proceeds with an optimistic sense of its progress. As time runs out, nobody admits how low the quality is for the chosen feature set. The deadline arrives, and you end up shipping a project with features that are 80% done.
- Eyes Open
  - Would have been better to accept the reality halfway through the project, cut some features, and end up shipping with a smaller number features 100% done. i.e. make tradeoffs among the 3 variables
- Alternate phrasing
  - It can be done on time. It can have all the features you want. It can be done well. Trade among these.
- Quotes
  - "Fast, Cheap, Good -- pick any two"
  - poster of above:
  - "You want it bad? We'll give it to you bad!"
- Conclusion
  - Take testing and milestones seriously to evaluate the current state or the project. Avoid denial. Make informed tradeoff decisions that balance the three variables.
- Note: People -- the 4th variable
  - You could argue that "people" is a fourth variable -- you could increase the number of people to do better with the other three variables.
  - I don't present it that way, since it is generally accepted that adding people to an in-progress project is often counterproductive. Fred Brooks quote: "adding people to a late software project makes it later." It is often true that adding people works poorly, but there are exceptions.



http://www.noise-to-signal.com/2003/04/pick_any_two.html

Quality Suffers

- In the features/time/quality tradeoff decisions, quality tends to suffer since it is intangible
- Challenger Explosion Example

- Features vs. deadline vs. safety tradeoff. One interpretation of the Challenger explosion is that "safety" was intangible vs. the other metrics, so it kept losing the little tradeoff decisions. Software quality can be hard to measure in the same way.

Denial Problems

Denial Schedule

- 1. Human nature to want to interpret and report things optimistically
  - Alice: I think there's huge monster behind this door.
  - Bob: Let's not look in there.
- 2. Quality of code is not immediately apparent
  - You have to actively test code it to see if it works.
  - With lack of time, it's very easy to not test hard. So the quality of the code is not really known. This path is less work and emotionally convenient.
- 3. When code components are really integrated and test run, it becomes undeniable what works and what does not.

Typical Slipping Schedule

- Everything looks fine until the last 20% of the schedule. Then there's a sickening realization that the project is much farther from completion than anyone thought.
- Somehow, for the middle part of the project, the team did not have a realistic picture of how far along things were.

At The End, It's Too Late

- You cannot make things proceed quickly at the end, no matter how badly you want to or how hard you push yourself, or how many people you have to throw at the problem
- Therefore, serious coding effort early in the process is the only way to have something workable at the end.
- Self Motivation
  - This is purely an issue of self-control and motivation -- everyone is motivated at the end, but it's too late. The skill is: work with that "night before it's due" intensity two weeks before it's due.
  - "The will to win means nothing without the will to prepare." - Juma Ikangaa, marathoner (also attributed to the basketball coach Bobby Knight)

Good News: Parallel Team Development

Parallel Development

- The focal technique for writing large projects -- modularity combined with teamwork.
- 3 people, 3 computers -- parallel
  - Writing and testing in parallel -- separate schedules, separate testing, separate bugs, separate computers -- not only are there three of you, but the logistics are much easier. A luxury most likely seen in the beginning
  - Depends very much on the design and separation between the components -- this is why abstraction and ADTs are stressed so much in cs106-cs107-cs108 -- optimize for "separateness" in the modules is the key for large projects.
- 3 people, 1 computer -- integrated
  - Always reduced to this eventually
  - Careful design to allows you enjoy the parallel model as long as possible
  - Careful design and testing allows the integrated phase will go well
- Point
  - Having a modular design that enables parallel development.
  - Someone should be able to work on module A without knowing much about module B

  - Someone should be able to mess with the implementation of module A without introducing bugs in the features of module B (imagine making changes in A just before you ship ... you really don't want new bugs popping up in B at that moment)

- Linux
  - How is it that all the people working on Linux are able to work independently while getting so much done? The person working on `gcc` vs. the person writing the `make` implementation? Careful abstractions and interfaces keep gcc and make quite separate from each other -- separate developers, separate release schedules, and separate bug databases. Most software components aren't as separate as gcc is from make (although perhaps they should be!).