The UNIVERSITY of VERMONT

Benjamin Fenster
CS-205 (Damon)
20 December 2004

Project Management

- I. People Management
  - a. Should be easiest of the three
  - b. Use common sense: Treat people like idiots; they'll act like idiots
  - c. Too many managers spend too much time alienating people.
- II. Process Management
  - a. Want to improve productivity
  - b. Existing techniques based on factory productivity (how many widgets produced)
  - c. Hard to measures code. What constitutes a line? Problems are unequal.
  - d. Huge variance in professional programmers in terms of lines of code produced *and* in terms of bugs per line of code
  - e. Mythical Man Month
    - i. Can't just put more people on the project and get faster results
    - ii. More people means more communication: $O(N^2)$
    - iii. Adding people to projects that are already late is the nightmare scenario – spend a month doing nothing but training new people and getting further behind
  - f. Tiger Team
    - i. Small group of really good programmers
    - ii. Hard to find six (say) really good people
    - iii. Some projects are so bulky this won't work anyway
    - iv. Some IT stuff requires less creativity so more programmers might be okay.
  - g. Process Improvement
    - i. Measure what you're doing now.
    - ii. Make changes, re-measure
    - iii. Keep the good changes, and repeat the procedure.
- III. Capability Maturity Model
  - a. Level 1. Initial
    - i. No recognizable process
    - ii. Just people doing tasks
  - b. Level 2. Repeatable: Undocumented, but consistent
  - c. Level 3. Defined: Documented
  - d. Level 4. Managed
  - e. Level 5. Optimizing
  - f. Moving from 1 to 2 is clearly advantageous.
  - g. To 3, usually good but a bit riskier.
  - h. To 4, more likely to end up worse off.
- IV. ISO 9000
  - a. Very specific detail about manufacturing parts, et cetera
  - b. Doesn't make a whole lot of sense in software
  - c. The same machine may make similar defects every day so it's advantageous to know which units came from which machine.
  - d. The same doesn't hold for programmers.
- V. PSP. Personal Software Process
  - a. At the individual level
  - b. Help yourself get better every day
- VI. Scheduling
  - a. Two ways you might need to approach scheduling
    - i. I know what we need. When can it be done?
    - ii. I know when it's due. What can we accomplish?
  - b. Can list tasks; find interdependencies, aggregate.
  - c. Schedule's only as good as time estimates
    - i. Typically from developers
    - ii. Safer to double estimates (developers underestimate)
  - d. Task Lists

i. Every class is a task, as is every interesting method.
ii. Common Problems. See [CS-205-2004-3-LECTURE-15:20]

e. Schedule Pressure: You have a week. Do it. Do it. (See *Starsky & Hutch*)

f. Working with Schedules
i. Have a list of tasks, times, dependencies
ii. Need to understand what that means about the project
iii. Gantt Chart
1. Chart of Task vs. Times
2. Grey boxes show expected remaining time
3. Black box shows percent completed ($^1/_3$, $^1/_2$)
4. Not meant to show dependencies

iv. Pert Chart
1. Developed by US Navy
2. Focused on dependencies
3. Still tasks vs. time but with arrows for dependencies – easy to see what depends on what.
4. Not good for progress check. Use a Gantt chart for that.
5. Each box is a task.
6. Can group boxes into rows by which developer will be performing the task, or by some other grouping.

v. Critical Path
1. What tasks *cannot* take longer without changing the end date?
2. Enough slippage of any one task will delay the project. What can't slip?
3. If one task takes longer or shorter, changes what's on the critical path?

g. Schedule Control
i. Manager does have some control
ii. Apply resources: Who does what and when? Want everybody busy all the time.
iii. Track schedule aggressively. Know how it's going every day (find slippage early)
iv. Want to make adjustments *before* someone starts a task.
v. May need to make new decisions about what can be cut without altering the date

h. Priority vs. Urgency
i. Urgency: How soon is it needed?
ii. Priority: How important is it?
iii. Disjoint!
iv. Preference priority over urgency
v. If it's urgent but low priority, may need to make a conscious decision to *not* work on it. Otherwise the inclination is to rush to get it done, possibly at the cost of something more important in the future.

VII. Risk Management
a. Manage risks too.
b. Schedule slippage
c. Technology failure
d. Problem is unsolvable
e. Bad implementation
f. Too buggy
g. Sources
i. Didn't understand the problem well
ii. Optimistic scheduling
iii. Personnel shortfall
iv. Changing requirements
v. External dependencies (e.g. new hardware)

h. Addressing Risks
i. Be realistic
ii. Aggressively track the schedule
iii. Prototype (aggressively)

VIII. Build Process

- a. Traditionally use overnight builds
- b. Process differs greatly from one group to another
- c. Developers modify current system
- d. Constant building: Changes the whole process
- e. Structure
  - i. Have a master directory – your directory
  - ii. Anything you're changing gets checked out; copied locally
  - iii. Really working with the whole *tree*
- f. Version Management
  - i. Need to organize code over time
  - ii. Source Safe (Microsoft), RCS (Freeware) – Most common
  - iii. RCS
    1. Difference in a single change is a *delta*
    2. One developer per file
    3. Check out file, make changes, check in
    4. If you *branch*, RCS will do the bulk of the merge automatically
    5. Creates RCS directory with version files.
    6. See [CS-205-2004-3-LECTURE-27]
- g. Configuration Management
  - i. CVS is the free system.
  - ii. Check out file or whole module.
  - iii. Distributed access (can work from home)

IX. Build Support

- a. make used broadly in UNIX, but als in Windows
- b. Some file (makefile) describes build behaviors
- c. Targets
  - i. Describe a goal
  - ii. Could be "compile a file"
  - iii. Could be "build, test, install the whole system"
- d. Makefile:
  - i. target : dependencies   action
  - ii. Make target → If any dependencies are newer than the target, perform the action
  - iii. Each dependency can be another target
  - iv. Action is any shell code
  - v. Common targets
    1. clean   Removes anything that can be reconstructed (.o, …)
    2. clobber Removes .c files too (check them out again afterward)
    3. source Create entire source tree (check out)
    4. devtree
    5. depend Updates dependencies in makefile

X. Bug Database

- a. Record every bug you've discovered along with a test that demonstrates it.
- b. Record what version introduced it and what version fixed it
- c. Associate related source files, affected features, …