Introsuction

Phil Gibbons
15-712 F15
Lecture 1

Waitlist Status

• As of noon today: 34 registered, 4 on waitlist

• Admittance priority: CSD PhD, ECE PhD, other SCS PhD, CSD Masters, ECE Masters, other Masters

• Priority among Masters students based on relevant courses taken (e.g., 15-213, 15-410) and grades obtained

• Last Fall, course capped at 24 students (project face-time limit)
  – I will admit qualified students off waitlist only if enrollment drops below 33
  – If you’re not going to take class, please drop so we know the real # of students

Today’s Topics

• Course Overview
  – No slides, just a walk through of the key points on the course webpages

• Discussion of 3 Wisdom Papers

The Mythical Man-Month

Fred Brooks 1975

• Why programming projects are hard to manage

  “Good cooking takes time. If you are made to wait, it is to serve you better, and to please you.”

• Tar Pit:
  – Program -> Programming Product (tested, documented) = 3x
  – Program -> Programming System (APIs, resource budget, testing) = 3x
  – Total = 9x programming time

• Woes: perform perfectly, authority below responsibility, dependent, debugging is tedious & slow to converge, program feels obsolete
Mythical Man-Month

- Optimism: Techniques of estimating time are poorly developed
- Fallaciously confuse effort (months) with progress
  - must consider communication overheads
- SW managers lack the courteous stubbornness of Antoine’s chef
  - false scheduling to match a patron’s date
- Schedule progress is poorly monitored

Brook’s Law: “Adding manpower to a late software project makes it later”

The Surgical Team

- Among experienced programmers, best are 10x productive and code is 5x faster/smaller
  - But small teams will take too long
- Team of 10: Surgeon, copilot, administrator, editor, 2 secretaries, program clerk, toolsmith, tester, language lawyer (performance hacks)
- Harder to scale up to larger teams

Aristocracy vs. Democracy

- Conceptual integrity is THE most important consideration in system design
- Ratio of function to conceptual complexity is the ultimate test of system design
- Division of labor between architecture (complete and detailed specification of the user interface) and implementation
  - what vs. how
  - can proceed somewhat in parallel

Second-System Effect

- An architect’s first work is apt to be spare and clean
- But second systems tend to go overboard
- Example: Static program overlays in OS/360 linkage editor
  - obsolete and slower than recompiling
Communication

- Specifications should be both formal definitions and prose definitions
  - don't use an implementation as specification
- Conferences, Courts, 2 implementations

Productivity & Size

- Interruptions while coding are bad
- Operating systems 3x slower to code than compilers, Compilers 3x slower than batch application programs
- Write two versions of each important routine: the quick and the squeezed
- Representation (data structure) is the essence of programming

Plan to Throw One Away

- ...you will anyway
- Plan the system for change (modular design)
- Have a Technical Cavalry at your disposal

Program Maintenance

- Cost of maintaining a widely-used program is typically 40% or more of the cost of developing it

"Program maintenance is an entropy-increasing process, and even its most skilful execution only delays the subsidence of the system into unfixable obsolescence"

The Whole and the Parts

- Program libraries: playpen, integration, version
- The most pernicious and subtle bugs are system bugs arising from mismatched assumptions made by authors of various components
- Use top-down design with stepwise refinement
- Many poor systems come from an attempt to salvage a bad basic design and patch it with all kinds of cosmetic relief
- Half as much code in scaffolding as in product
Hatching a Catastrophe

• How does a project get to be a year late?
  ...One day at a time
• During the activity, overestimates of duration come steadily down as the activity proceeds
• Underestimates do not change significantly during the activity until about 3 weeks before the scheduled completion
• Do critical path planning analysis (PERT chart)
• Self-document programs: comment the source code

You and Your Research

Richard Hamming 1986

• Hamming distance
• Hamming codes (first error correcting codes)
• Turing Award winner 1968
• "The purpose of computing is insight not numbers"

Q: Why do so few scientists make significant contributions and so many are forgotten in the long run?

How to be a Great Scientist

• “Luck favors the prepared mind” – Pasteur
• As teenagers, they had independent thoughts and the courage to pursue them
• Key Characteristic: Courage
• Do best work when they are young professionals
  – When you are famous it is hard to work on small problems
  – Fail to plant the acorns from which the mighty oaks grow
  – The IAS at Princeton has ruined more good scientists than any institution has created

How to be a Great Scientist

• People are often the most productive when working conditions are bad
• Most great scientists have tremendous drive
  – must be intelligently applied
• Knowledge and productivity are like compound interest
• Great scientists tolerate ambiguity well
• Are completely committed to their problem
  – keep your subconscious starved so it has to work on your problem
How to be a Great Scientist

- What are the important problems in your field?
  - and must have plan of attack

- Great Thoughts Time

- The great scientists, when an opportunity opens up, get after it and they pursue it

- He who works with the door open gets all kinds of interruptions, but he occasionally gets clues as to what the world is and what might be important

- Never again solve an isolated problem except as characteristic of a class

- Do your job in such a fashion that others can build on it

How to be a Great Scientist

- Need to sell your work, via good writing, formal talks, and informal talks

- Make talks be more big picture

- Is the effort to be a great scientist worth it?

- Personality defects such as wanting total control, refusing to conform to dress norms, fighting the system rather than take advantage of it, ego, anger, negativity
  - Let someone else change the system

- Know yourself, your strengths and weaknesses, and your bad faults

How to be a Great Scientist

- Should get into a new field every 7 years

- The bigger the institutional scope of the vision, the higher in management you need to be

- In the long-haul, books that leave out what’s not essential will be most valued

- Do library work to find what the problems are

- Refuse to look at any answers until you’ve thought the problem through carefully how you would do it, how you could slightly change the problem to be the correct one

- Choose the right people to bounce ideas off of

The Rise of Worse is Better

Richard Gabriel 1991

- MIT/Stanford style of design: “the right thing”
  - Simplicity in interface 1st, implementation 2nd
  - Correctness in all observable aspects required
  - Consistency
  - Completeness: cover as many important situations as is practical

- Unix/C style: “worse is better”
  - Simplicity in implementation 1st, interface 2nd
  - Correctness, but simplicity trumps correctness
  - Consistency is nice to have
  - Completeness is lowest priority
**Worse-is-better is Better for SW**

- Worse-is-better has better survival characteristics than the-right-thing

- Unix and C are the ultimate computer viruses
  - Simple structures, easy to port, required few machine resources to run, provide 50-80% of what you want
  - Programmer conditioned to sacrifice some safety, convenience, and hassle to get good performance and modest resource use
  - First gain acceptance, condition users to expect less, later improved to almost the right thing
  - Forces large systems to reuse components; no big monolithic system

---

**To Read/Summarize for Friday**

- “Hints for Computer System Design”
  Butler Lampson 1983

- “End-to-End Arguments in System Design”
  Jerome Saltzer, David Reed, David Clark 1984

- “The UNIX Time-Sharing System”
  Dennis Ritchie and Ken Thompson 1974

**Optional Further Reading:**

- “Programming Semantics for Multiprogrammed Computations”
  Jack Dennis and Earl Van Horn 1966