CSE584: Software Engineering  
Lecture 1: Introduction & Overview  

David Notkin  
Computer Science & Engineering  
University of Washington  
http://www.cs.washington.edu/education/courses/584/  

Outline  

• Intent and overview of course  
• Software engineering overview  
  – Stuff you already know, but it’s important to lay it out so we are working from the same page  
• Notkin’s top 10 “insights”  
  – My goal is to lay out my prejudices and views, to increase your understanding of the intent of the course  
• Overview of course work and administrivia  

Introductions  

• Very useful for me (and you)  
  – What do you do?  
  – What do you want from the class?  
  – What are the most serious software engineering problems you face?  
• But time consuming, so we’ll do it electronically  
  – Through the email list (cse584@cs.washington.edu)  
  – Distributed to the entire class  

But I do want some basics  

• What companies do you work for?  
• What is your general responsibility?  
  – Development, testing, maintenance, other?  
• Take a couple of minutes at each site to gather these data  
  – I’ll handle the UW site  
  – The person whose last name comes first alphabetically handles the other sites  
• Announce when you’re ready  

Interaction  

• I like to have interaction with students during class, especially 584  
  – You have tons of key insights in your head  
  – It’s boring just listening to me  
  · Especially in the evening & during a long class  
• Try just interrupting me; if that doesn’t work, we’ll try something else  
• Remind me to repeat questions, since it’s often hard to hear them at other sites  

Your undergraduate experience?  

• How many of you took an undergraduate software engineering course?  
• Did any of you think it was good?  
• What, specifically, was particularly good or bad about it?  

This is my guess about your answers  

For non-UW grads, that is!
Intent of course

- Most of you have jobs engineering software
  - I don’t (and I never really have)
- So, what can I teach you?
  - Convey the state-of-the-art
    - Especially in areas in which you don’t usually work
  - Better understand best and worst practices
  - Consider differences in software engineering of different kinds of software
- You provide the context and experience
- Meeting and talking to each other is key

Lots of differences among you

- You have a lot in common
  - Undergrad degree in CS or related field
  - Significant experience in the field
  - You’re really smart
- You also have a lot of differences
  - Development vs. testing
  - Desktop vs. real-time
  - Different company cultures
  - ...and much, much more
- This in part will be why some material in the course will resonate with you, while other material won’t

My metric for success

I will consider this a successful course if, over the course of the next year or so, you approach some specific problem you face differently because of the course

- Maybe from readings
- Maybe from discussions with other students
- Maybe from assignments
- Maybe even from lecture

Another key intent

- There is general agreement that
  - Research in software engineering doesn’t have enough influence on industrial practice
  - And much in industry could improve
- Why is this true?
  - What can academia do to improve the situation?
  - What can industry do to improve the situation?
- By the way, I believe that this perception is not entirely accurate
  - But it’s still a crucial issue for lots of reasons

Possible impediments

- Lack of communication
  - Industry doesn’t listen to academia
  - Academia doesn’t understand industrial problems
- Academic tools often support programming languages not commonly used in industry

Other possible impediments?

- In groups of 3 or 4, list some other possible impediments
- In 3 minutes, we’ll gather a set of suggestions

GO! STOP!
Tichy’s main impediment

- Walter Tichy has claimed that the major impediment is the lack of “experiments” in CS research
- http://www.ipd.ira.uka.de/~tichy/
- I have lots of reactions, including
  - I don’t think industry, as a rule, finds this to be a (the) major impediment
  - We do experimentation, in a different style
  - Evaluation is difficult in software engineering, so we must be creative
  - This is an example of science envy

Software is increasing critical to society

and it’s getting bigger and more complex

Absolute sizes

- ATM (my 3 year-old)
- B-2 (Tao on my shoulders)
- NT5.0 (Statue of Liberty from base)
- 50 lines per page
- Double sided
- 500 pages/ream (2 inches)

How I spend my time

- The Great Pyramid of Giza is 481’
- The Kingdome was 250’
- The Colossus of Rhodes is 110’
- The Eiffel Tower is 1033’
- The Graduate Reading Room in Suzzallo is 65’
- A 747 is 36’ to the top of the tail
- The Brooklyn Bridge is 135’ above the water
- Titanic’s height from keel to bridge is 104’
- The EE1 building is about 90’

Delivered source lines per person

- Common estimates are that a person can deliver about 1000 source lines per year
- Including documentation, scaffolding, etc.
- Independent of the programming language
- Yeah, you do better
- Obviously, most complex systems require many people to build
- Even an order of magnitude increase doesn’t eliminate the need for coordination

Inherent & accidental complexity

- Brooks distinguishes these kinds of software complexity
  - We cannot hope to reduce the inherent complexity
  - We can hope to reduce the accidental complexity
- Some (much?) of the inherent complexity comes from the incredible breadth of software we build
- That said, it’s not always easy to distinguish between these kinds of complexity
“The Software Crisis”
- We’ve been in the midst of a “software crisis” ever since the 1968 NATO meeting
  - crisis — (1) an unstable situation of extreme danger or difficulty; (2) a crucial stage or turning point in the course of something [WordNet]
  - I was 13, and many of you weren’t born yet
- We cannot produce or maintain high-quality software at reasonable price and on schedule
  - Gibb’s Scientific American article [in your course pack]
  - “Software systems are like cathedrals; first we build them and then we pray” — S. Redwine

Some classic “crisis” issues
- Relative cost of hardware/software
  - Where’s Moore’s Law for software?
- Low productivity
- “Wrong” products
- Poor quality
  - Importance depends on the domain
- Constant maintenance
  - “If it doesn’t change, it becomes useless”
- Technology transfer is (too) slow

Notkin’s view—“mostly hogwash”
- Given the context, we do pretty well
  - We surely can, should and must improve
- Some so-called software “failures” are not
  - They are often primarily management errors
    - Ariane, Denver airport, U.S. air traffic control, etc.
    - Interesting recent article in the Wall Street Journal on Australia’s and New Zealand’s success in air traffic control
    - Read comp.risks
- In some areas, we may indeed have a looming crisis
  - Safety-critical real-time embedded systems
  - Y2K wasn’t

Software engineering is a “wicked problem”
- Cannot be easily defined so that all stakeholders agree on the problem to solve
- Require complex judgments about the level of abstraction at which to define the problem
- Have no clear stopping rules
- Have better or worse solutions, not right and wrong ones
- Have no objective measure of success
- Require iteration — every trial counts
- Have no given alternative solutions — these must be discovered
- Often have strong moral, political or professional dimensions

Other problems
- Lack of well-understood representations of software [Brooks] makes customer and engineer interactions hard
- Relatively young field
- Software intangibility is deceptive

Law XXIII, Norman Augustine [Wulf]
- “Software is like entropy. It is difficult to grasp, weighs nothing, and obeys the second law of thermodynamics; i.e., it always increases.”

S. Buckingham Shum
http://kmi.open.ac.uk/people/sbs/org-knowledge/aikm97/sbs-paper2.html
## Dominant discipline

As the size of the software system grows, the key discipline changes.

<table>
<thead>
<tr>
<th>Code Size</th>
<th>Discipline</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^3$</td>
<td>Mathematics</td>
</tr>
<tr>
<td>$10^4$</td>
<td>Science</td>
</tr>
<tr>
<td>$10^5$</td>
<td>Engineering</td>
</tr>
<tr>
<td>$10^6$</td>
<td>Social Science</td>
</tr>
<tr>
<td>$10^7$</td>
<td>Politics</td>
</tr>
<tr>
<td>$10^8$</td>
<td>??</td>
</tr>
</tbody>
</table>

[Stu Feldman, thru 107]

## Notkin’s Top 10 Observations

- **About software engineering**
  - With apologies and appreciation to many unnamed souls
- **I’d appreciate help revising this list over the quarter**
- **And, again, the intent of this is to convey, now, many of my prejudices**
  - You’re not required to share them, but you’ll understand more because I’m being explicit about (most of) them

### 1. Don’t assume similarity among software systems

- Does (and should) the reliability of a nuclear power plant shutdown system tell us much about the reliability of an educational game program?
- Does (and should) the design of a sorting algorithm tell us much about the design of an event-based GUI?
- So, assume differences until proven otherwise: not doing so causes a tremendous amount of confusion in the degree of applicability of different research approaches, tools, etc.

### 2. Intellectual tools dominate software tools in importance

- How you think is more important than the notations, tools, etc. that you use
- **Ex: Information hiding is a key design principle**
  - Interface mechanisms can enforce information hiding decisions but cannot help one make the decisions
- **Ex: The notion of design patterns is more important than languages that let you encode them**

### 3. Analogies to “real” engineering are fun but risky

- One reason is because of the incredible rate of change in hardware and software technology
  - Wulf: what if the melting point of iron changed by a factor of two every 18 months?
- Another is that software seems to be constrained by few physical laws
- But I’ll make them anyway, I’m sure
  - And you will, too

### Aside: should software engineers be licensed?

- You may have heard about this issue
  - For example, Texas now requires (under some conditions) that software engineers be licensed as professional engineers
- It’s an incredibly complex issue
  - Technically, socially, politically and legally
  - I’d be happy to discuss my views on this with individuals (including on the mailing list), but I won’t spend time in class on it
- **BTW, I am strongly opposed to licensing software engineers for the foreseeable future**
4. Estimating benefits is easier than estimating costs

- “If only everyone only built software my way, it’d be great” is a common misrepresentation
  - Ex: The formal methods community is just starting to understand this
- But at the same time, estimating the costs and the benefits is extremely hard, leaving us without a good way to figure out what to do

5. Programming languages ensure properties distant from the ones we want

- Programming languages can help a lot, but they can’t solve the "software engineering" problem
- Ex: Contravariant type checking (such as in ML) has significant benefits, but regardless, it doesn’t eliminate all errors in ML programs
  - And covariant typing, with its flaws, may be useful in some situations

6. The total software lifecycle cost will always be 100%

- Software development and maintenance will always cost too much
- Software managers will always bitch and moan
- Software engineering researchers will always have jobs
- Software engineers will always have jobs

7. Software engineering is engineering

- Although software engineering draws heavily on mathematics, cognitive psychology, management, etc., it is engineering in the sense that we produce things that people use
  - It’s not mathematics, nor cognitive psychology, nor management (nor etc.)
  - Nor logical poetry (cf. the Michael Jackson video we’ll see later in the quarter)
- If somebody is talking about engineering software without ever mentioning "software", run away

8. Tradeoffs are key, but we’re not very good at them

- Getting something for nothing is great, but it isn’t usually possible
- We almost always choose in favor of hard criteria (e.g., performance) over soft criteria (e.g., extensibility)
  - This makes sense, both practically and theoretically
  - Brooks’ Golden Rule doesn’t really work
  - But the situation leaves us up a creek a lot of the time
- Maybe we’re about to get better at this as the cost of people continues to grow
  - But I doubt it

9. It’s good to (re)read anything by Brooks, Jackson & Parnas

- “A classic is something everyone wants to have read, but nobody wants to read.” [Mark Twain]
- It’s more important to read their works than to read the latest glossy rag or modern book on the latest fad
- Really
10. Researcher ⇔ Practitioner

- Software engineering researchers should have a bit of the practitioner in them, and software engineering practitioners should have a bit of the researcher in them.
- At the end of the quarter, I hope that I’ll have more understanding of practice, and you’ll have more understanding of the research world.

Overview—five topics

- Requirements and specification
- Design
- Evolution (maintenance, reverse engineering, reengineering)
- Analyses and tools (static and dynamic)
- Quality assurance and testing
- Yes, there is some overlap
  - I reserve the right to completely change my mind about the order and exactly what is covered!

What’s omitted? Lots

- Metrics and measurement – Some in QA
- CASE – Some in evolution and tools
- Software process – CMM, ISO 9000, etc.
- Specific methodologies
- [UX]ML
- Software engineering for specific domains (real-time, the web, etc.)
- What else?

Requirements & specification (2 lectures)

- Formal methods
  - State-based, algebraic, model-based
  - Model checking
- Problem and domain analysis
  - Problem frames, use-case, collaborations, etc.
- Highlight: A Michael Jackson video

Design (2 lectures)

- Classic topics
  - Information hiding
  - Layered systems
  - Event-based designs (implicit invocation)
- Neo-modern design
  - Limitations of classic information hiding
  - Design patterns
  - Software architecture
  - Frameworks

Evolution (2 lectures)

- Why software must change
- How and why software structure degrades
- Approaches to reducing structural degradation
- Problem-program mapping
- Program understanding, comprehension, summarization
Analyses and Tools (2 lectures)

- Static analyses
  - Type checkers
  - Extended type checkers
- Dynamic analyses
  - Profiling
  - Memory tools
  - Inferring invariants

Quality assurance (1 lecture)

- Verification vs. validation
- Testing
  - White box, black box, etc.
- Reliability
- Safety (maybe)

Anything else?

Overview of course work

- Four assignments, each of a different form
  - A standard homework, a paper distilling research in an area, an assessment of a research prototype tool, etc.
  - All turned in electronically; each worth 23% of the grade
- The other 8% will represent your interaction in lecture and (more importantly) on the mailing list
  - Discussion of papers, of lectures, and of other software engineering issues on your mind
  - This is especially important for a distance learning class
  - It’s the best way to learn from each other
  - You are responsible for pushing the discussion threads, although the TA and I will participate

Grading: Let’s make a deal

- If you focus on the material and don’t get compulsive about grading …
- … then I will focus on the material and not get compulsive about grades

Goodnight

- And don’t forget to buy those course packs