CSE584: Software Engineering
Lecture 1 (April 1, 1997)

David Notkin
Dept. of Computer Science & Engineering
University of Washington
www.cs.washington.edu/homes/notkin

Intent of course

- Most of you have jobs engineering software
  - I don’t
- So, what can I teach you?
  - Convey the state-of-the-art
  - Better understand best and worst practices
  - Consider differences in software engineering of different kinds of software
- You provide the context and experience

Overview—four topics

- Design
- Evolution (maintenance, reverse engineering, reengineering)
- Requirements and specification
- Quality assurance and testing

(Plus tonight’s overview of software engineering)

What’s omitted? Lots

- Metrics and measurement
  - Some in QA
- Tools & environments (CASE)
  - Some in evolution and QA (and a bit tonight)
- Software process
  - CMM, ISO 9000, etc.
- Specific methodologies
- What else?

Design (2 lectures)

- 1st lecture—classic topics
  - Information hiding
  - Layered systems
  - Event-based designs (implicit invocation)
- 2nd lecture—neo-modern design
  - Limitations of classic information hiding
  - Design patterns
  - Software architecture
<table>
<thead>
<tr>
<th>Evolution (2 lectures)</th>
<th>Requirements (2 lectures)</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Why software must change</td>
<td></td>
</tr>
<tr>
<td>● How and why software structure degrades</td>
<td></td>
</tr>
<tr>
<td>● Approaches to reducing structural degradation</td>
<td></td>
</tr>
<tr>
<td>● Problem-program mapping</td>
<td></td>
</tr>
<tr>
<td>● Program understanding, comprehension, summarization</td>
<td></td>
</tr>
<tr>
<td>● Domain analysis</td>
<td></td>
</tr>
<tr>
<td>● Requirements elicitation</td>
<td></td>
</tr>
<tr>
<td>● Formal methods</td>
<td></td>
</tr>
<tr>
<td>– State-based, algebraic, model-based</td>
<td></td>
</tr>
<tr>
<td>● Use-case, collaborations, etc.</td>
<td></td>
</tr>
<tr>
<td>● Analysis techniques</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quality assurance (2 lectures)</th>
<th>Anything else you want to cover?</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Verification vs. validation</td>
<td></td>
</tr>
<tr>
<td>● Formal verification</td>
<td></td>
</tr>
<tr>
<td>● Testing</td>
<td></td>
</tr>
<tr>
<td>– White box, black box, etc.</td>
<td></td>
</tr>
<tr>
<td>● Reliability</td>
<td></td>
</tr>
<tr>
<td>● Safety</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Overview of course work</th>
<th>Assignment 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Five assignments—working in pairs permitted</td>
<td></td>
</tr>
<tr>
<td>– Introduction + one per focus topic</td>
<td></td>
</tr>
<tr>
<td>● Group (2-3) or individual report</td>
<td></td>
</tr>
<tr>
<td>– Groups focus on defining state-of-the-art in one of the four focus topics</td>
<td></td>
</tr>
<tr>
<td>– Individuals negotiate reports (or projects) with me</td>
<td></td>
</tr>
<tr>
<td>● Final exam (6/10/97, Sieg 422)</td>
<td></td>
</tr>
<tr>
<td>● Intentions</td>
<td></td>
</tr>
<tr>
<td>● Expectations</td>
<td></td>
</tr>
</tbody>
</table>
Michael Jackson, ICSE-17

- Deep thinker & clear speaker
- More focused on requirements topic
  - But some general software engineering insights as well
- Better than listening to me
- We can stop the video and discuss issues as they arise

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

Number 1

- We make a huge mistake by assuming a priori that there similarity among software systems
  - So, assume differences until proven otherwise

Number 2

- Intellectual tools still dominate mechanical tools in importance
  - How you think is more important than the notations, tools, etc. that you use

Number 3

- Analogies to other engineering disciplines are attractive but generally fall apart quickly because of the incredible rate of change in hardware and software technology.
  - But I’ll make them anyway, I’m sure

Number 4

- It is often too easy to estimate the benefits of a “better” approach to engineering software without assessing its costs
  - “If only everyone only built software my way, it’d be great,” just doesn’t work.
Number 5
◆ The properties that programming languages can ensure are still distant from the properties we require software systems to have
  – Programming languages can help a lot, but they can't solve the "software engineering" problem

Number 6
◆ The total software lifecycle cost will always be 100%
  – Software development and maintenance will always cost too much
  – Software engineering researchers will always have jobs

Number 7
◆ Software engineering draws on mathematics, cognitive psychology, management, etc., but it is engineering, not mathematics, nor cognitive psychology, nor management (nor etc.)
  – If somebody is talking about software without ever mentioning "software", run away

Number 8
◆ Tradeoffs are at the heart of software engineering, but we're not very good at making tradeoffs yet
  – Getting something for nothing is great, but it isn't usually possible
  – At the least, it takes a great designer

Number 9
◆ It's always good to read and re-read anything written by Brooks, Jackson, and Parnas
  – Don't fall into Mark Twain's trap:
    » "A classic is something everyone wants to have read, but nobody wants to read."

Number 10
◆ Software engineering researchers must have a bit of the practitioner in them, and software engineering practitioners must have a bit of the researcher in them
Software is critical to society

- Economically important
- Essential for running more enterprises
- Key part of most complex systems
- Essential for designing many engineering products

(Old) sample code sizes [Jon Jacky]

<table>
<thead>
<tr>
<th>System</th>
<th>Code Size (LOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bar code scanners</td>
<td>10-50KLOC</td>
</tr>
<tr>
<td>4-speed transmissions</td>
<td>20KLOC</td>
</tr>
<tr>
<td>ATC ground system</td>
<td>130KLOC</td>
</tr>
<tr>
<td>Automated teller machine</td>
<td>600KLOC</td>
</tr>
<tr>
<td>Call router</td>
<td>2.1MLOC</td>
</tr>
<tr>
<td>B-2 Stealth bomber</td>
<td>3.5MLOC</td>
</tr>
<tr>
<td>Seawolf submarine combat</td>
<td>3.6MLOC</td>
</tr>
<tr>
<td>NT 4.0</td>
<td>6MLOC + 4MLOC scaffolding</td>
</tr>
</tbody>
</table>

Delivered source lines per person

- Common estimates are that a person can deliver about 1000 source lines per year
  - Including documentation, scaffolding, etc.
- 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

“The Software Crisis”

- We’ve been in the midst of a “software crisis” ever since the 1968 NATO meeting
  - We are unable to produce or maintain high-quality software at reasonable price and on schedule
    - Wayt’s *Scientific American* article
    - “Software systems are like cathedrals; first we build them and they we pray” —Redwine

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 management errors (Ariane, Denver airport, etc.)
- In some areas, in particular safety-critical real-time embedded systems, we may indeed have a looming crisis
Some “crisis” issues

- Relative cost of hardware/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 slow

Why is it hard?

- There is no single reason software engineering is hard—it’s a “wicked problem”
- Lack of well-understood representations of software [Brooks] makes customer and engineer interactions hard
- Relatively young field
- Software intangibility is deceptive

Dominant discipline

- As the size of the software system grows, the key discipline changes
- Due to Stu Feldman

<table>
<thead>
<tr>
<th>Code Size</th>
<th>Discipline</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^9$</td>
<td>Mathematics</td>
</tr>
<tr>
<td>$10^8$</td>
<td>Science</td>
</tr>
<tr>
<td>$10^6$</td>
<td>Engineering</td>
</tr>
<tr>
<td>$10^5$</td>
<td>Social Science</td>
</tr>
<tr>
<td>$10^4$</td>
<td>Politics</td>
</tr>
</tbody>
</table>

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.”

“Is software engineering” engineering?