CSE584: Software Engineering

Lecture 1 (April 1, 1997)

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

UW CSE
CSE584: Software Engineering

Notkin (c) 1997 1

Lecture 1, Outline [approximate minutes]

- Introductions—names, professional backgrounds, professional literature that you read, interest in course [10]
- Intent and overview of course [15]
- Overview of course work (assignments, projects, etc.) [10]
- Videotape clips of Michael Jackson's keynote talk from ICSE-17 [45]
- Break [10]
- Software engineering overview [55]
- Group discussion— CASE: past, present & future [15]
- Administrivia and slop [10 minutes]
  - Reschedule lecture, 22 April 1997 (Passover)?
  - Reschedule lecture, 20 May 1997 (ICSE-97)

Notkin (c) 1997 2

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

Notkin (c) 1997 3

Overview—four topics

- Design
- Evolution (maintenance, reverse engineering, reengineering)
- Requirements and specification
- Quality assurance and testing
- (Plus tonight's overview of software engineering)

Notkin (c) 1997 4

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?

Notkin (c) 1997 5

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

Notkin (c) 1997 6

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

Notkin (c) 1997 7

Requirements (2 lectures)

- Domain analysis
- Requirements elicitation
- Formal methods
  - State-based, algebraic, model-based
- Use-case, collaborations, etc.
- Analysis techniques

Notkin (c) 1997 8

Quality assurance (2 lectures)

- Verification vs. validation
- Formal verification
- Testing
  - White box, black box, etc.
- Reliability
- Safety

Notkin (c) 1997 9

Anything else you want to cover?

Notkin (c) 1997 10

Overview of course work

- Five assignments—working in pairs permitted
  - Introduction + one per focus topic
- Group (2-3) or individual report
  - Groups focus on defining state-of-the-art in one of the four focus topics
  - Individuals negotiate reports (or projects) with me
- Final exam (6/10/97, Sieg 422)

Notkin (c) 1997 11

Assignment 1

- Intentions
- Expectations

Notkin (c) 1997 12

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

UW CSE CSE584: Software Engineering Notkin (c) 1997 13

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

UW CSE CSE584: Software Engineering Notkin (c) 1997 14

Number 1

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

UW CSE CSE584: Software Engineering Notkin (c) 1997 15

Number 2

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

UW CSE CSE584: Software Engineering Notkin (c) 1997 16

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

UW CSE CSE584: Software Engineering Notkin (c) 1997 17

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.

UW CSE CSE584: Software Engineering Notkin (c) 1997 18

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]

Bar code scanners	10-50KLOC
4-speed transmissions	20KLOC
ATC ground system	130KLOC
Automated teller machine	600KLOC
Call router	2.1MLOC
B-2 Stealth bomber	3.5MLOC
Seawolf submarine combat	3.6MLOC
NT 4.0	6MLOC + 4MLOC scaffolding

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

SE <> PL

-
-
-
-
-
-

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

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.”

Dominant discipline

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

Code Size	*Discipline*
10^3	Mathematics
10^4	Science
10^5	Engineering
10^6	Social Science
10^7	Politics

“Is software engineering” engineering?