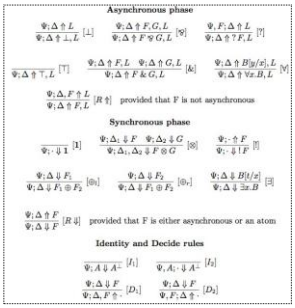
CSE P503: Principles of Software Engineering

David Notkin
Spring 2009

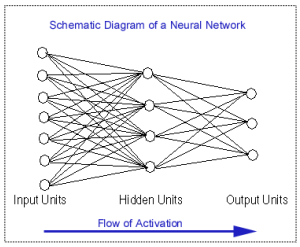
Weeks 1-3: formal specifications

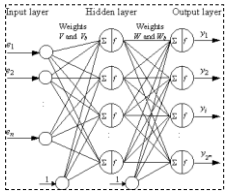


Start with Hoare logic and work to linear logic and modal logic

Ax. 1. $P(\varphi) \wedge \Box \forall x[\varphi(x) \to \psi(x)] \to P(\psi)$
Ax. 2. $P(\neg\varphi) \leftrightarrow \neg P(\varphi)$
Th. 1. $P(\varphi) \to \Diamond\, \exists x\, [\varphi(x)]$
Df. 1. $G(x) \iff \forall\varphi[P(\varphi) \to \varphi(x)]$
Ax. 3. $P(G)$
Th. 2. $\Diamond\, \exists x\, G(x)$
Df. 2. $\varphi \text{ ess } x \iff \varphi(x) \wedge \forall\psi\{\psi(x) \to \Box\, \forall x[\varphi(x) \to \psi(x)]\}$
Ax. 4. $P(\varphi) \to \Box\, P(\varphi)$
Th. 3. $G(x) \to G \text{ ess } x$
Df. 3. $E(x) \iff \forall\varphi[\varphi \text{ ess } x \to \Box\, \exists x\, \varphi(x)]$
Ax. 5. $P(E)$
Th. 4. $\Box\, \exists x\, G[x]$

Weeks 4-7: using neural nets





Understand basics of neural nets, practice with various approaches to weighting them, apply to software engineering problems

Weeks 8-10: software reliability

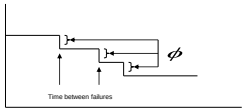
$$E_r(T) = \frac{E_0}{I_0} - E_c(T) \qquad \hat{N} = \frac{\sum_{i=1}^{n} f_i}{(1-\exp(-\hat{b}t_n)}$$

$$R(t,T) = e^{-C\left(\frac{E_0}{I_0} - E_c(T)\right)\cdot t}$$

$$\frac{t_n \exp(-\hat{b}t_n)\sum_{i=1}^{n} f_i}{(1-\exp(-\hat{b}t_n)} = \sum_{i=1}^{n} \frac{f_i(t_i \exp(-\hat{b}t_i) - t_{i-1}\exp(-\hat{b}t_{i-1})}{\exp(-\hat{b}t_{i-1}) - \exp(-\hat{b}t_i)}$$

$$MTTF = \int_0^{\infty} R(t)dt$$

Error models, fault/failure models, non-homogeneous distributions, etc.



April Fool! GOTCHA!

- Oh, that was yesterday?
- No problem, just an off-by-one error!

Material on the previous slides taken without attribution but with apologies to many sites and people

Facts

- Collectively and individually, you have designed, developed, tested, shipped and maintained orders of magnitude more software than I have
- Collectively and individually, you continue to make design decisions, write code, test code, fix bugs, etc. on a daily basis; I don't
- Few of you are aware of much ongoing research in software engineering; I am
- Few of you are able to separate quickly the good from the bad in software engineering research; I am good (although imperfect) at this

Course goals

- To expose you to key approaches in software engineering research, with the hope that one or more of them can help you in your daily work – perhaps immediately, perhaps in the longer term
- Without ignoring your day-to-day issues, try to look deeper into the issues of engineering quality software than day-to-day pressures usually allow
- To let you delve into some specific research areas that interest you
- To increase your ability to communicate with software engineering researchers and other software engineers

Your problems?

- What problems – technical or non-technical – do you find the most serious in designing, developing, maintaining, and shipping software and/or products that have significant software components?

Groups of 3-4

Some from former PMP students

- Lack of open communication
- Inability to prepare for and adjust to unexpected changes
- Nailing down interfaces
- Software development does not get much recognition as an art
- Quality is always what loses in the battle between development and management
- Methods for mitigating bugs early in the software process are not well known or accepted
- Servicing software and maintaining backwards compatibility
- Lack of scheduled design time
- Lack of proper specifications
- Lack of proper documentation for old code
- Lack of processes that allow for writing, building and testing the code and then releasing it such that customers are not adversely affected
- Designing software so that it is very easy to test
- Loss of knowledge when people move on

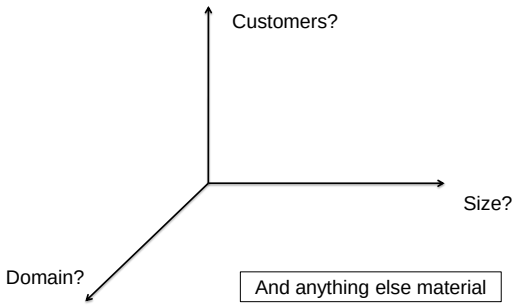
Your academic background?

- Undergraduate degree in computer science or computer engineering?
- Undergraduate degree in something else?
- Undergraduate course in software engineering?
- Other academic programs or degrees?

Software engineering course?

- If you took an undergraduate software engineering course, what was the structure?
  - Small teams, large teams?
  - Assigned project, self-defined project, no project?
  - Full lifecycle, early lifecycle, late lifecycle?
- What did you like or not like?
- Was any material relevant to your (first) job? If so, what was it?

Your organization?



Topics you'd like to see? Not see?

Some of those mentioned by former PMP students

- Measuring "quality" objectively
- Important results from research (especially quantitatively evaluated)
- Deep underlying theory that's normally underappreciated or ignored by practitioners
- Project management, managing project scope
- SOA
- UML
- …

Groups of 3-4

Quantitatively evaluated results

- Thought experiment:
  - *Without having to demonstrate that a result is accurate, state an imaginable "quantitative result" that would drive your daily work more effectively*

Scrum development is…

- … better than Extreme Programming.
- … better than Extreme Programming in 23% of projects .
- … results in 41% fewer bugs than does Extreme Programming.
- … better than Extreme Programming in 59% of projects that have at most 30 software developers.
- … better than Extreme Programming in 61% of projects that have largely inexperienced software developers.
- … better than Extreme Programming in 52% of projects in which at least 15% of the software developers have come from Engineering schools.

- OK, you try.

Enough about you…

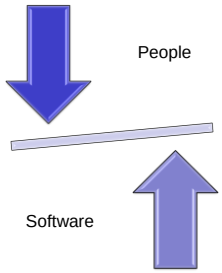
- Brown (1977), Carnegie Mellon (1984)
- UW since 1984, department chair 2001-06
- Advised/co-advised 19 PhD students
- Sabbaticals in Japan (1990-91), Israel/Japan (1997-98), Sweden (2006-07)
- Program chair 1st ACM SIGSOFT Symposium on the Foundations of Software Engineering (1993)
- Program co-chair 17th International Conference on Software Engineering (1995)
- ACM SIGSOFT chair (1997-2001)
- *ACM Transactions on Software Engineering and Methodology* editor-in-chief (2007-)
- CRA (Computing Research Association) board (2005-)
- …

Questions, comments, anecdotes…

- I won't learn much if you keep quiet during lecture and electronically outside of class
  - And yes, it's all about me! ☺
- You won't learn as much either
  - Research shows that in lecture people have a relatively short attention span; maybe 15-20 minutes near the beginning of a lecture, dropping to just a few minutes later on
  - The attention span "clock" can be reset by questions and other non-"yadda yadda" interludes
- Help me continue to learn about "the customer" – all of you! – so that we all take full advantage of your experience

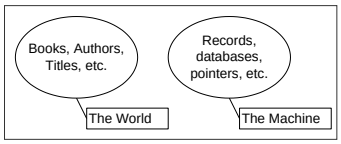
Perspectives and biases #1

- Barry Boehm distinguishes between building the system right and building the right system
- Michael Jackson distinguishes between requirements in the world and programs that define the machine
- Manny Lehman and Les Belady identify the feedback loop between the users and a program



Michael Jackson

- Requirements in the application domain
- Program (machine) has an effect in the application domain
- The mapping is inherently imperfect



Serious confusions abound when distinctions like these are forgotten or ignored

- Things in the world that are not represented in the machine: e.g., book sequels, pseudonyms, anonymous books
- Things in the machine that don't represent anything in the world: e.g., null pointers, deleting a record, back pointers

Perspectives and biases #2

- "Software crisis" coined in 1968 at 1st NATO Software Engineering Conference
- Software projects are too expensive, too buggy, too late, cancelled too often, …
- Cyber-physical projects are too late, fail too often, etc. due to software

- Therac-25
- Mars Polar Lander
- Mars Climate Orbiter
- Ariane
- Denver Airport
- Vancouver Stock Market
- Concon
- …

Quite simply, software and software engineering suck

Resolved: software [engineering] sucks

Pro: all of you

- Prepare points in small groups [~5 minutes]

Con: me

"pro" nor "con"

- We need to assess and consider value
- We need to work towards defining realistic bounds on absolute costs, relative costs and intrinsic costs
- We need to consider all dimensions of engineering, including the physical components and the users themselves

If software is really a "crisis", give us another one!

Making software the "fall guy" will not help us solve the important and hard problems we face

Perspectives and biases #3

- Engineering (including software) is design under constraints
- You are intimately, although at times implicitly, aware of your constraints: customer needs, shipping deadlines, resource limitations (memory, power, money, etc.), compatibility, reward structure, organizational culture, and much more…
- I do not know your constraints, which makes it at least hard to know which approaches and techniques can be effectively applied in your context

A consequence of varied constraints

- There is no *single right way to engineer software*: no best programming language, design method, software process, testing approach, team structure, etc.
- This does not imply that every approach is good under some constraints
- Nor does it suggest that there are no consistent themes across effective approaches

- "I have the uncomfortable feeling that others are making a religion out of [removing gotos], as if the conceptual problems of programming could be solved by a single trick, by a simple form of coding discipline!" [E. Dijkstra]
- "Don't get your method advice from a method enthusiast. The best advice comes from people who care more about your problem than about their solution." [M. Jackson]

Perspectives and biases #4

- Is software engineering really engineering? Can it be? Should it be? Will it be?
  - Maturity and relevance of the field?
  - Continuing change?
  - Dominant discipline?
  - Kind of design?
  - Physical constraints?
  - Moore's Law for software?

Maturity and relevance

- For better or for worse, the software industry became relevant incredibly quickly (on an historical basis)
- The mashup of development, research, startups, and more appears to be different from other "engineering" fields (on an historical basis)
- Open question: to what degree, if any, are the problems faced by the software field a matter of its immaturity? If this is indeed an issue, are there ways to cause us to mature more quickly?

"All useful programs undergo continuing change": Belady and Lehman

- A significant amount of "software maintenance" addresses changes for which roughly analogous changes would be considered non-routine in most other fields
  - Augmenting a radio to include a television
  - Adding floors to skyscrapers, lanes to bridges
  - Accommodating new aircraft at airports
  - Adding Cyrillic-based languages to European Union documents
  - Adding support to a browser for an entirely type of interaction (e.g., digital pens)
  - Scaling software systems by an order of magnitude (pick your dimension)
  - Supporting the web in a desktop productivity suite
  - Adding support for Asian languages to a tool

Dominant discipline: Stu Feldman

10^3 Lines of Code	Mathematics
10^4 LOC	Science
10^5 LOC	Engineering
10^6 LOC	Social Science
10^7 LOC	Politics
10^8 LOC, 10^9 LOC, …	???, ???, …

Kinds of design

- Routine vs. innovative design
  - Designing a C compiler for a new DSP chip
  - Designing the first WYSIWYG editor
- Standardized vs. non-standardized design
  - Automobile design is standardized: the designers know virtually everything about the context in which the automobile will be used (expected passenger weights, what kind of roads will be encountered, etc.)
  - Bridge design is non-standardized: the designers must understand the specific location in which the bridge will be built (the length of the span, the kind of soil, the expected traffic, etc.)
- These lead to fundamentally different design spaces – where does software fit?

Software and physical laws

- Physical systems are constrained by largely well-known and well-understood laws of physics
- Many of these laws rely on notions of continuity, where small changes in an input generally lead to a small change in the output
- Continuous mathematics is a powerful model for these systems
- Software instead works in a discrete world, where small changes in an input often lead to discontinuous changes in the output
  - Discrete math must face enormous state spaces
- Failure modes differ – failure of physical components vs. design flaws
- "Software is like entropy. It is difficult to grasp, weighs nothing, and obeys the second law of thermodynamics; i.e., it always increases." [Norman Augustine]

Moore's Law?

"… exponentially improved hardware does not necessarily imply exponentially improved software performance to go with it. The productivity of software developers most assuredly does not increase exponentially with the improvement in hardware, but by most measures has increased only slowly and fitfully over the decades." [Wikipedia,"Software: breaking the law"]

- The performance of software *and* of software developers is compared to transistors on an integrated circuit
- What human activity has matched the growth of Moore's Law? The productivity of hardware designers?
- What other technology has matched the growth of Moore's Law? Batteries? Displays?

Is it really engineering?

- Overall, I believe that software is – at least at present – sufficiently different from physical materials that software engineering should be considered to be largely distinct from classic engineering disciplines
- Many of the approaches that try to make software engineering more like engineering seem to do so by trying to beat the "soft" out of "software" – but isn't that precisely its potential and its power?

Perspectives and biases #5

- Cyber-physical systems are even harder to think about
- Co-design often pushes hard stuff into software (after all, it's "just" software) – which naturally makes the software more complicated: and complex stuff is more likely to have flaws because it's complex
- Co-design freezes non-software parts early, so software must fix any problems in those parts (after all, software is "soft")
- Software comes last, so it's often blamed

Two cyber-physical examples

Therac-25

Death from lethal radiation doses

- Code wasn't independently reviewed
- Software wasn't considered during reliability modeling
- A physical interlock was removed: it had masked defects in earlier models
- The software could not verify that sensors were working correctly
- Experienced operators could enable a race condition – but testing was done with inexperienced operators
- Overflow weakened error checking

Mars Polar Lander

$120M crash

- "…the most likely cause of the failure of the mission was a software error that mistakenly identified the vibration caused by the deployment of the lander's legs as being caused by the vehicle touching down on the Martian surface, resulting in the vehicle's descent engines being cut off while it was still 40 meters above the surface, rather than on touchdown as planned." [Wikipedia]

Just because different software would make a difference doesn't necessarily mean it was a software problem *per se*

Again…

- Knee-jerk reactions to software are bad for everybody – we need more accuracy, more honesty
- As software professionals, we need to be articulate about what we do well and what we do poorly and what we know and what we don't know
- The root cause is not always the same as the direct cause

Perspectives and biases #6

- Crucial judgments about software are made by humans informed by technical assessments – this will not change
- The technical assessments may be wrong
- The technical assessments may be insufficient
- The assumptions underlying the technical assessments may be wrong
- The assumptions the humans make about the technical assessments may be wrong
- The judgments of the humans may be wrong

Perspectives and biases #7

- It's a matter of human confidence
  - evidence
  - assumptions
  - argument

Capturing confidence

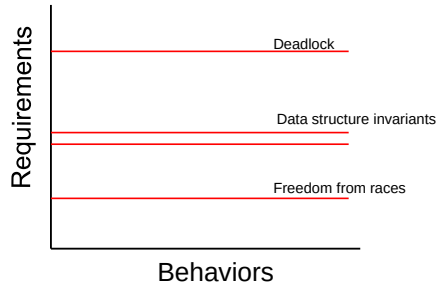
- Matt Dwyer and colleagues put this notion of confidence in terms of "sources of unsoundness"
  - We need to know the degree of unsoundness
  - That is, we need to know what we know, and what we don't know
- Bev Littlewood and colleagues use Bayesian Belief Networks to assess confidence levels
  - "How much will confidence about a system's safety increase if I add a verification argument to a statistical testing argument?"
  - Plausible BBN-based answers include
    - reducing doubt by1/3 and
    - supportive verification lessening confidence from testing alone

Dwyer: Coverage

- Nobody (rational) believes that one technique will "do it all": a suite of techniques will be required
- How do we know that these techniques
  - cover the breadth of software requirements?
  - cover the totality of program behavior?
- That is, how do we know that
  - every desired property (correctness, performance, reliability, security, …) is achieved in
  - every possible execution?

Sample across executions

Sample across requirements



Possible topics: TBD

Requirements and specifications

- More software systems fail because they don't meet the needs of their users than because they aren't implemented properly
- A brief history in proving programs correct
  - An expected panacea for software that didn't pan out
  - But has provided some benefits
- A look at formal specifications, with a focus on two forms
  - Model-based specifications (Z) – we'll come back to automatic analysis of specifications like these later on
  - Overview of state machine based specifications – including automatic analysis using model checking
- A brief overview of requirements engineering issues

Design

- Basic issues in design, including some historical background
  - Well-understood techniques such as information hiding, layering, event-based techniques
- More recent issues in design
  - Aspect oriented approaches
  - Architecture, patterns, frameworks

Evolution

- The objective is to use an existing code base as an *asset*
- Basic background
- Approaches to change
  - Reverse engineering
  - Visualization
  - Software summarization
- Change as a first-class notion
- Augmenting Dwyer's view with change
- Longitudinal analysis

Analysis and tools

- Tools and analysis
- The analysis part might be close to the specification topics covered earlier in the quarter, but the focus will be much, much closer to the source code
- Static vs. dynamic analysis
- Underlying representations
- Example tools

Quality assurance/testing

- What do we know, and when do we know it?
- Building confidence over time

Mining software repositories

- "Research is now proceeding to uncover the ways in which mining [software] repositories can help to understand software development, to support predictions about software development, and to plan various aspects of software projects." [MSR 2007 web page]
  - Broadly defined to include code, defect databases, version control information, programmer communications, etc.
- Underlying premise: we believe there is something – actually, a lot of things – that can be learned from studying these repositories
- But it presents a paradox – if we think most software is low quality, how can we learn by studying the repositories?

Final examination

- By University rule, an instructor is allowed to dispense with a final examination at the scheduled time (6:30-8:20PM, June 11, 2009) with unanimous consent of the class
- If you prefer to have a final examination for the entire class, you *must* let me know by the 6:00PM before the second lecture (April 9, 2009)

Four assignments

1. Essay
2. A secondary research report on an approved topic based on significant reading of various pertinent papers and materials
   - These scholarly reports provide information about the topic and your analysis of it, complete with citations, open questions, etc.
3. Non-tool based assignment
4. Tool-based assignment (probably in Daniel Jackson's alloy system)

- Unless there's a final, these are 25% each
- The research report and the tool-based assignments may be done in groups up to three people

First assignment: essay

- Due two weeks (minus a couple of hours) from now
- A 5-10 page articulate, well-reasoned essay, with appropriate citations about one of three topics
- Post your essays on the wiki
  - 1/5 of your grade for the assignment will be based on timely comments on essays by the other students

Topic A

- Consider the 1968 and 1969 NATO Software Engineering Conferences. Characterize issues that (a) have been solved, (b) are no longer material, and (c) are still pertinent but remain unsolved. Also identify current technologies, methodologies, etc. (if any) that are argued to address the pertinent-but-not-yet-solved issues.

Topic B

- Consider three or four "software disasters" not discussed in class. Describe each of them with some care and provide a thoughtful analysis of the core causes of each disaster. Pick disasters for which there is a non-trivial analysis. Conclude the essay with an assessment of the way these disasters are generally presented in comparison to your own analysis.

Topic C

- Consider the SWEBOK Guide, Chapter 1, "Introduction to the Guide" (found in several formats at the site) and "An Assessment of Software Engineering Body of Knowledge Efforts", A Report to the ACM Council (May 2000, by Notkin, Gorlick and Shaw).
- Thoughtfully argue that the SWE Body of Knowledge guide is or is not an appropriate basis for the licensing of software engineers.

Remember

- Stay tuned to the web page and the wiki
- I will try to be in my office (CSE542, 206-685-3798) for the hour or so before each class
  - I am happy to take email and phone calls and to make appointments

Before you leave: one-minute paper

- Most important point made in class tonight?
- Unanswered questions you still have?
- Any recommended mid-course corrections?