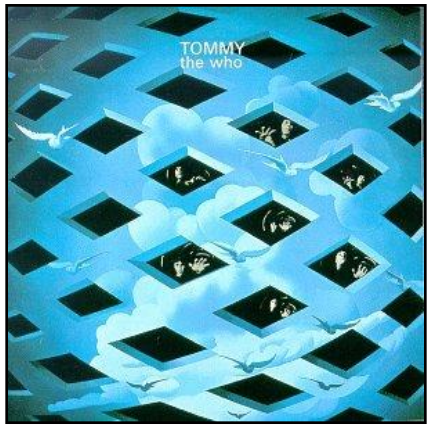
CSE P503: Principles of Software Engineering

David Notkin
Autumn 2007

[Example from Michael Jackson]

Principle 0: Establish communication



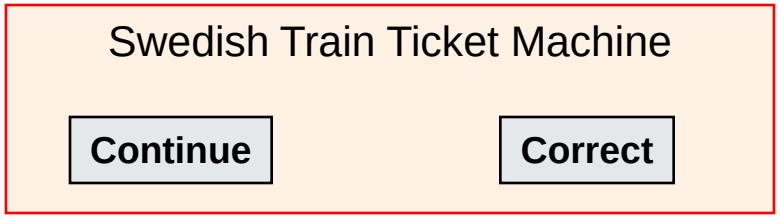
Tommy, can you hear me?

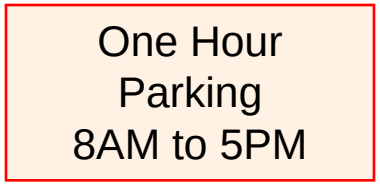
Do you copy?

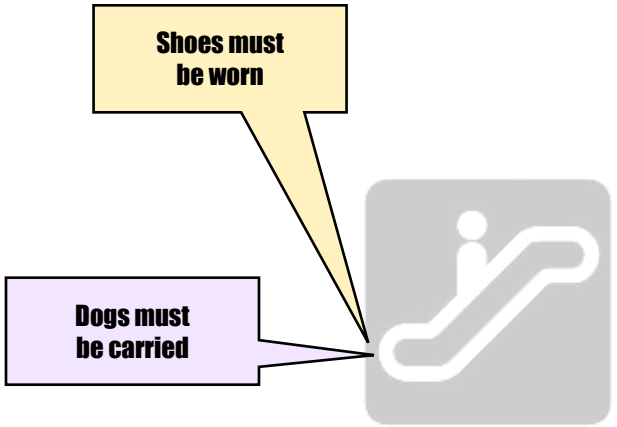
Can you hear me now?

Communication isn't easy

- "America and England are two countries divided by a common language."
  - Wilde or Shaw or Churchill
- Nouns, verbs, tenses, moods, cultural context, and much more







Formalism to the rescue!

- Since natural language is the culprit, instead use mathematical languages, inference rules, logics, etc. that are suitable to automated manipulation

  $\forall$**i,j:1..N, i < j** $\Rightarrow$ **A[i]** $\leq$ **A[j]**

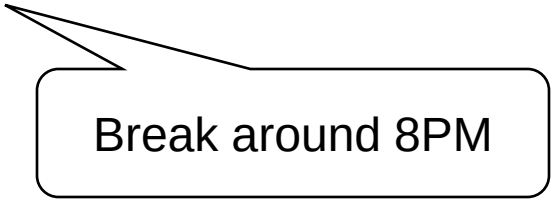
- A logical formula (post-condition) for describing a sort program
- After a sort program executes, there are no two unordered elements in the array

But consider the program…

- ```
  for i := 1 to N do
      A[i] := i;
  end
  ```
- It satisfies the post-condition, which it "shouldn't"
- Must add another conjunct: **`A′= permutation(A)`**
- Formalization can help in some situations – but it doesn't and can't eliminate communication problems
- Building the system right vs. building the right system (Barry Boehm)
  - Formalization tends to help more with the former than the latter
  ```

Interlude

- More principles
  - A set of views on how to think (or at least how I think) about software engineering
- Lecture plans for the course
- Expected work
- Miscellaneous wrap-up
- A one-minute paper
  - What was the most important point made in class?
  - What unanswered question do you still have?
  - What mid-course correction, if any, would you suggest already?



Principle #1: know the customer

- Did you take an undergraduate software engineering course…?
- Are there topics that you would like to see covered/not covered in the course?
- Do you write/modify code and/or other artifacts on a regular basis?
- What problems – technical and other – do you find the most serious in developing, maintaining, and shipping software and/or products that have significant software components?
- Anything else that might help me prepare a better course?

Answers: sample size O(20)

- Undergraduate background: widely varies
- “Don’t make us write more code!”
- Most deal with code regularly, some deal with other artifacts regularly

Answers: to cover or not to cover

- Leveraging user feedback and usability study for product design
- 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
- State of the art in software process, agile methods
- Open source development
- Global software development
- Design, design patterns, design for maintainability, reusability, testing
- Effective QA
- How long should software “live”?
- Service Oriented Architectures
- Standards and interoperability
- …

- Process
- Requirements specifications
- Testing
- UML

Answers: problems you face

- Lack of open communication
- Ability 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
- …

Facts

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

So, the course objectives are…

- 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)
- To let you delve into some specific research areas that interest you
- To help you do your job in a more thoughtful and more systematic way, even in the absence of specific approaches that you pick up
- To increase your ability to communicate with software engineering researchers and other software engineers

Caveats

- Overall, we'll focus on technical approaches more than on management and process approaches
- The material, while by no means mostly mine, is surely Notkin-centric – this is at least as much a question of omission as inclusion

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

Principle #2: calling it a crisis does not make it so

- "Software crisis" coined by Friedrich Bauer at the 1st NATO Software Engineering Conference (1968)
- Ill-defined term that usually includes (note the similarity to your list of problems)
  - Software projects are late and over-budget
  - Software doesn't meet user needs
  - Software quality is low
  - Software is hard to manage
  - Software is hard maintain
  - Software engineering isn't "real" engineering
  - Software improves much more slowly than hardware (we have no Moore's Law)
- W. Gibbs, "Software's Chronic Crisis," *Scientific American* (1994)

“Crisis”

- “a vitally important or decisive stage in the progress of anything; a turning-point; also, a state of affairs in which a decisive change for better or worse is imminent” [OED]
- “an unstable or crucial time or state of affairs in which a decisive change is impending; *especially* **:** one with the distinct possibility of a highly undesirable outcome” [Merriam-Webster]
  - Cuban missile crisis
  - Subprime lending crisis
  - AIDS crisis
  - …

- Is decisive change imminent in software engineering?
- Is there a distinct possibility of a highly undesirable outcome for software engineering?

Decisive change is not imminent: Fred Brooks, No Silver Bullet (1987)

"The familiar software project … is capable of becoming a monster of missed schedules, blown budgets, and flawed products. So we hear desperate cries for a silver bullet – something to make software costs drop as rapidly as computer hardware costs do.

"But, as we look to the horizon of a decade hence, we see no silver bullet. There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity. …

"Although we see no startling breakthroughs – and indeed, I believe such to be inconsistent with the nature of software – many encouraging innovations are under way. A disciplined, consistent effort to develop, propagate, and exploit these innovations should indeed yield an order of-magnitude improvement. There is no royal road, but there is a road."

An undesirable outcome is unlikely

- Indeed, software projects do fail, are often costly and late, etc.
- At the same time, software provides enormous value
  - Software is at the heart virtually all complex systems, is core infrastructure for complex organizations, …
  - "... dramatic improvements in computing power and communication and information technology appear to have been a major force behind [the higher U.S. growth rate in productivity] output." [A. Greenspan, 1998]
  - Hoover's reports (2007): "Computer software production in the US involves about 50,000 companies with combined annual revenue of about $180 billion" [Note: this doesn't included related industries like software services]
- If this is a crisis, let's create another one!
  - We don't want AIDS, we don't want broad defaulting of loans, and we didn't want missiles in Cuba
  - … but we do want software!

Woody Allen, *Annie Hall*

- "The food in this place is really terrible."
  "Yes, and such small portions."

- This captures much of the confusion about software: it's broadly believed to be of low quality, but there is a voracious appetite for it
- *So, even if it's not a "crisis," we need to and we can do better, perhaps much better*

Principle #3: Engineering 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
- Committing to a "best approach" can be limiting

- "Please don't fall into the trap of believing that I am terribly dogmatical about [the goto statement]. I have the uncomfortable feeling that others are making a religion out of it, 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, Principle of Dispassionate Methodology]

An example

- The passionate methodologists include many (but not all) advocates of typed programming languages and formal methods
  - They have pushed research, and in some cases industry, forward in important and impressive ways
- At the same time, I believe that they work under a questionable assumption, which is largely captured by Dijkstra:
  - “Program testing can be used to show the presence of bugs, but never to show their absence!” [1970]
  - In other words, proving properties across all possible executions is good, while sampling across executions is inherently flawed
- Very roughly, this pushed the computing research community towards static and away from dynamic approaches – in a sense, an entire class of approaches was discounted wholesale
  - Need I note that most of industry uses dynamic approaches, often exclusively?

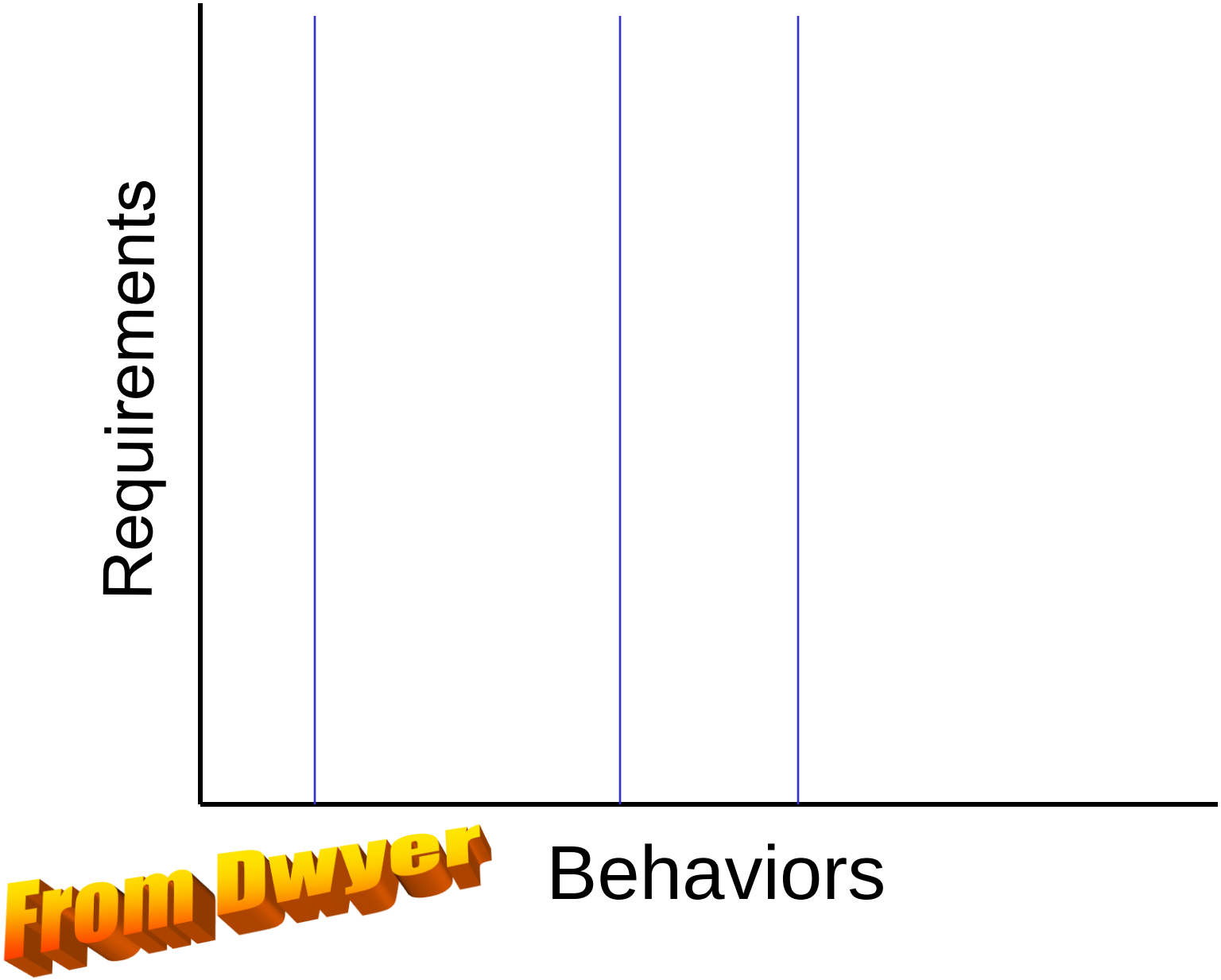
Sources of unsoundness: Dwyer et al. (2007)

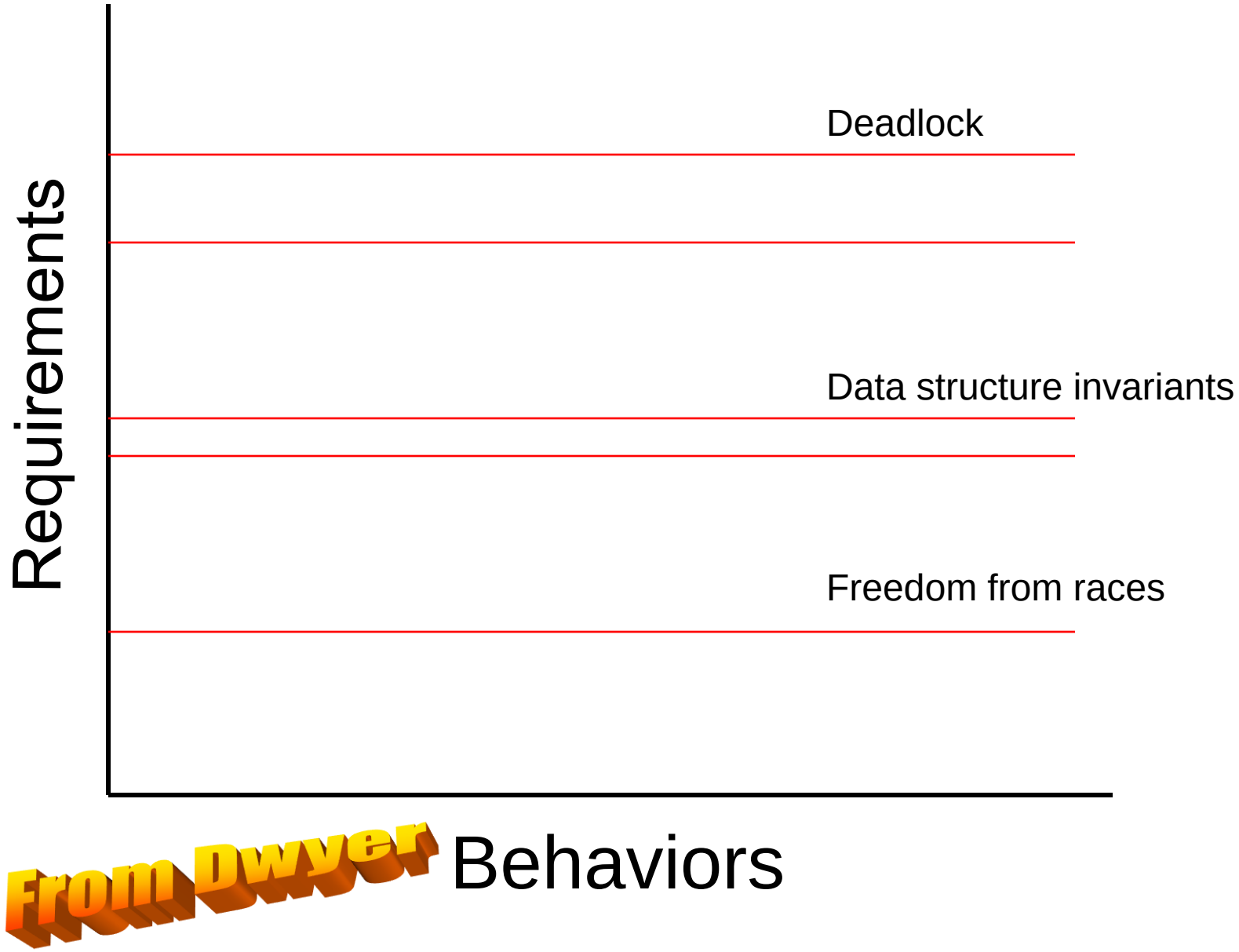
- Matt Dwyer's talk at ICSE 2007 put much of this issue in perspective: in my words, he argues that it's *all* sampling
- Dynamic techniques sample across executions (behaviors)
- Static techniques sample across properties (requirements)
- What we need to know is the degree of unsoundness
  - That is, we need to know what we know, and what we don't know
- The following few slides are from Dwyer's talk

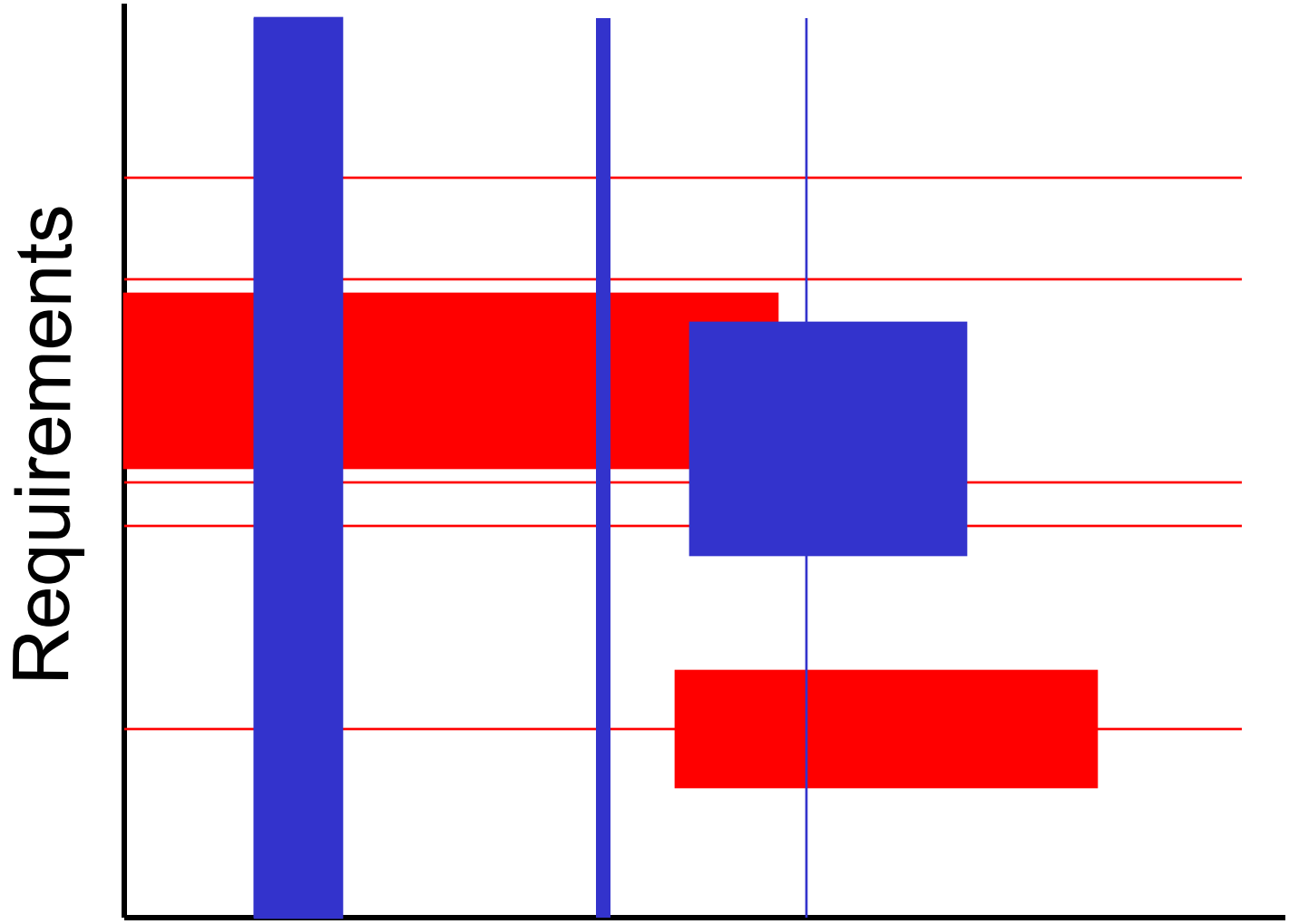
Its about Coverage

- Nobody believes that one technique will “do it all”
- A suite of techniques will be required
- If techniques involve sampling how do we know
  - They cover the breadth of software requirements
  - They cover the totality of program behavior

A unified theory of behavioral coverage is needed


Requirements
Behaviors
From Dwyer


Requirements
Deadlock
Data structure invariants
Freedom from races
From Dwyer
Behaviors



From Dwyer

Principle #4: software is different (until and unless proven otherwise)

- Dominant discipline
- Moore's Law for software?
- Specialized vs. standardized design
- Continuing change
- Maturity and relevance of the field
- Software structures not subject to physical laws

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, …	???, ???, …

Moore's Law

- "…the number of transistors on a chip doubles about every two years" (Moore, 1965)
- There's no Moore's Law for … automobile or aircraft design, power engineering, medical technology, or (almost?) anything other than chip design – that is, chip design is the singularity, not software
- It is at least arguable, too, that hardware improvements that focus primarily on speed force contribute to software's complexity

Design space complexity: Specialized vs. standardized [Michael Jackson]

- Designing both automobiles and bridges requires *specialized* knowledge
- 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.
- But bridge design is *not 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.
- This leads to fundamentally different design spaces

Software design

- Software design is widely and wildly non-standardized
  - Yes, it is also specialized
- Figuring out what the user wants and needs is hard and is almost always part of the job; for most software systems, this goes far beyond designing a bridge for a specific location
  - Again, Boehm's distinction between building the system right vs. building the right system
- A classic exception is some classes of compilers
  - The PQCC project at CMU (Wulf et al., 1980) led to the formation of Tartan Laboratories, which was acquired by TI (1996) primarily to construct C compilers for DSPs – in essence, this became standardized
  - Jackson suggests that "compiler engineering" (and such) might make sense, in contrast to "software engineering"

“All useful programs undergo continuing change”: Belady and Lehman (1976)

- 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 new document type (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

“So damned relevant”: Bill Wulf

- 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?

Software and physical laws

- Physical systems (bridges, automobiles, buildings, etc.) 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 – and under well-defined conditions – 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
  - The state spaces are enormous, and we have far less mathematics on which to rely (although we're making some progress)
- 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]

Principle #5: software is often the fall guy

- Software development is condemned in no uncertain terms
- It's a software crisis, right?

❖ "The Standish Group [1994] research shows a staggering 31.1% of projects will be canceled before they ever get completed. Further results indicate 52.7% of projects will cost 189% of their original estimates. The cost of these failures and overruns are just the tip of the proverbial iceberg. The lost opportunity costs are not measurable, but could easily be in the trillions of dollars. One just has to look to the City of Denver to realize the extent of this problem. The failure to produce reliable software to handle baggage at the new Denver airport is costing the city $1.1 million per day"

❖ "For nine months, this Gulliver has been held captive by Lilliputians-errors in the software that controls its automated baggage system." [Gibbs, "Software's Chronic Crisis", Scientific American, 1994]

Let's pile on…

Jussi Koskinen ● http://www.cs.jyu.fi/~koskinen/smcosts.htm

- Annual software maintenance cost in USA has been estimated to be more than **$70 billion** (Sutherland, 1995; Edelstein, 1993).
- E.g. in USA, the federal government alone spent about **$8.38 billion** during a 5-year period to the Y2K-bug corrections.
- At company-level, e.g. Nokia Inc. used about **$90 million** for preventive Y2K-bug corrections.

Was it (only) software?

- The Denver system was 14 times bigger than SFO's, and the first to serve an entire airport
  [Myerson, New York Times, 1994]
- And it was the first to
  - have carts slow down but not stop for bags
  - use networked desktops rather than a mainframe
  - use radio links
  - handle oversized bags … and more
- Much of this didn't go smoothly – but software took essentially all of the blame

Therac-25 – radiation therapy machine
Leveson and Turner (1993)

- A handful of patients received potentially lethal radiation doses when the machine before being targeted – several died
- Contributing problems included
  - The code wasn't independently reviewed
  - The software wasn't considered during reliability modeling
  - A physical interlock present in earlier models was removed; the interlock had masked software defects in those earlier models
  - The software could not verify that sensors were working correctly
  - Experienced operators could enable a dangerous race condition – evidently testing was done with inexperienced operators
  - A flag variable was set by incrementing it, with overflow weakening the error checking
- Yes, software was partially culpable, but so was the co-design, the management and more – but at least popularly, this is viewed as a tragic example of a software failure

Mars Polar Lander (1999)

- "…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]
- But an equally reasonable view is that it was an error in the sensors that mistakenly generated the vibration before touchdown

Why? And why care? In part …

- Co-design often pushes hard stuff into software – so it's harder (after all, it's "just" software)
  - And complex stuff is more likely to have flaws
- 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

- 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
  - The root cause is not always the same as the direct cause

What about software-only systems?

- Are these arguments only valid for complex systems that include software?
- Some of them are, perhaps
- But given the non-standardization of software design, the difficulty of identifying and addressing users' needs, the difficulty imposed by the discrete nature of software, and more, simple assertions that software is to blame is naïve and will not likely help us improve
- Once again, one must remember that in general one should not look at costs alone, but rather look at value – and as an industry, software clearly provides enormous value

Principle #6: we need better ways to think about the cost of software maintenance

- Not only is software absolutely costly, as seen above
- But some aspects, such as software maintenance/evolution are relatively too costly, too

Evolution dominates software costs

Jussi Koskinen ● *ibid*

Year	Proportion of software maintenance costs	Definition	Reference
2000	>90%	Software cost devoted to system maintenance & evolution / total software costs	Erlikh (2000)
1993	75%	Software maintenance / information system budget (in Fortune 1000 companies)	Eastwood (1993)
1990	>90%	Software cost devoted to system maintenance & evolution / total software costs	Moad (1990)
1990	60-70%	Software maintenance / total management information systems (MIS) operating budgets	Huff (1990)
1988	60-70%	Software maintenance / total management information systems (MIS) operating budgets	Port (1988)
1984	65-75%	Effort spent on software maintenance / total available software engineering effort.	McKee (1984)
1981	>50%	Staff time spent on maintenance / total time (in 487 organizations)	Lientz & Swanson (1981)
1979	67%	Maintenance costs / total software costs	Zelkowitz *et al.* (1979)

Thought experiments

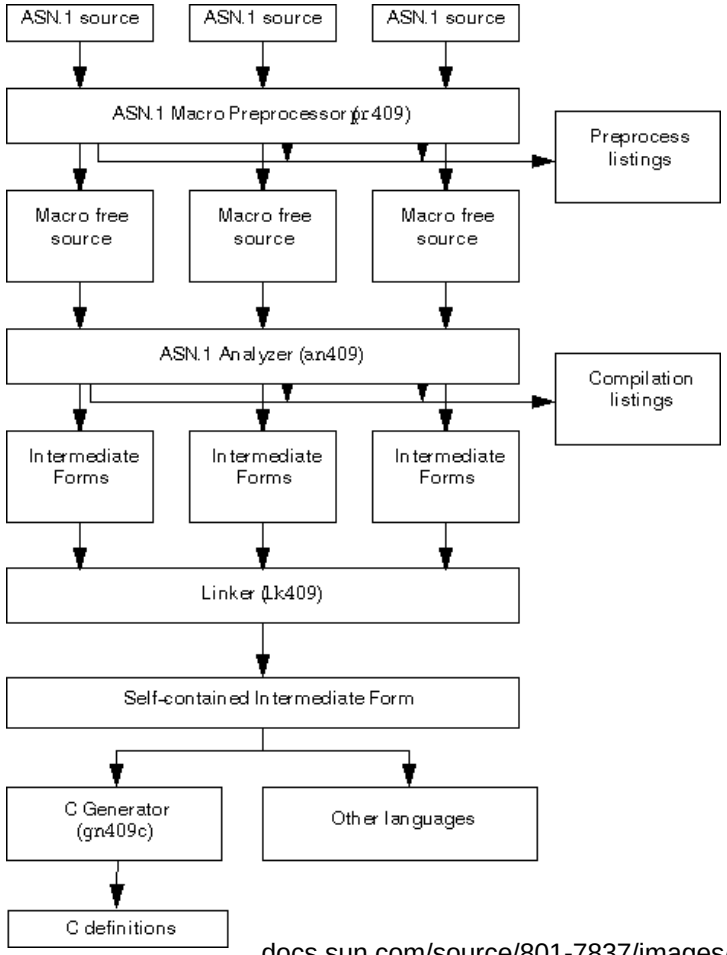
- What percentages would demonstrate “success” in software evolution? Is higher better or worse? Is there a numeric goal?
  - For extra credit: consider that software testing accounts for roughly half the total cost of software development [Beizer 90]
- What absolute numbers would demonstrate “success” in software engineering in general and software evolution in particular?

My rule of thumb: proportional cost

- What seems to be a genuine concern is when the cost of changing the implementation of an existing system is out of proportion with the apparent cost of the change in the requirements
  - Y2K – essentially no change in the requirements
  - Converting a single-threaded program to a distributed, reliable system over a wide-area LAN
- This is very hard to make operational
  - How to think about apparent costs? Apparent costs to whom?
  - Realistic measures of actual costs?

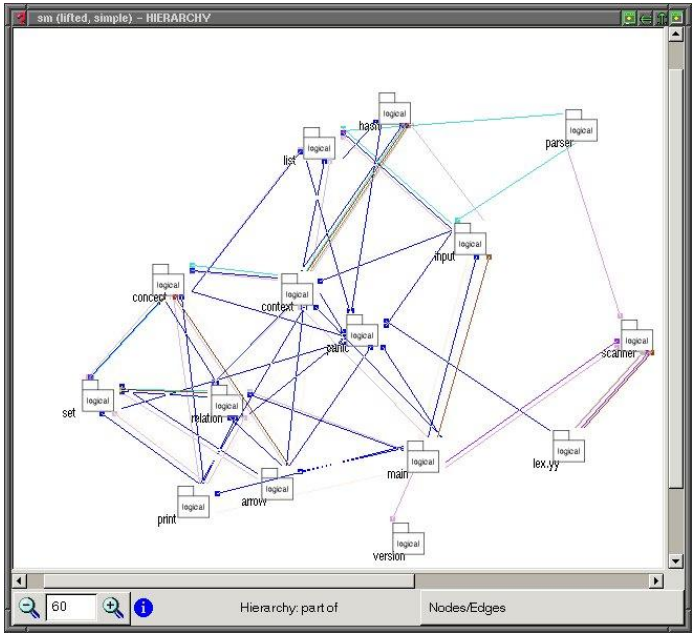
Principle #7: we lie

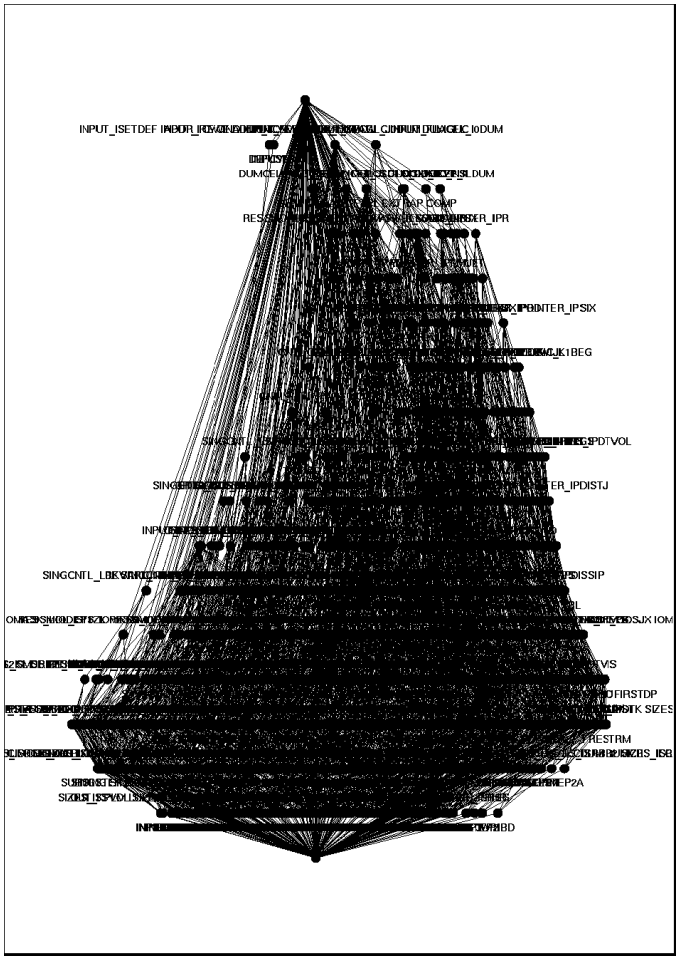
- We lie about software structure to our students, to our non-technical managers, etc.
- Ex: Should it be a surprise when a non-technical manager believes that a change will be simple, based on design diagrams?
- This directly affects the notion of “apparent costs” in the previous slide



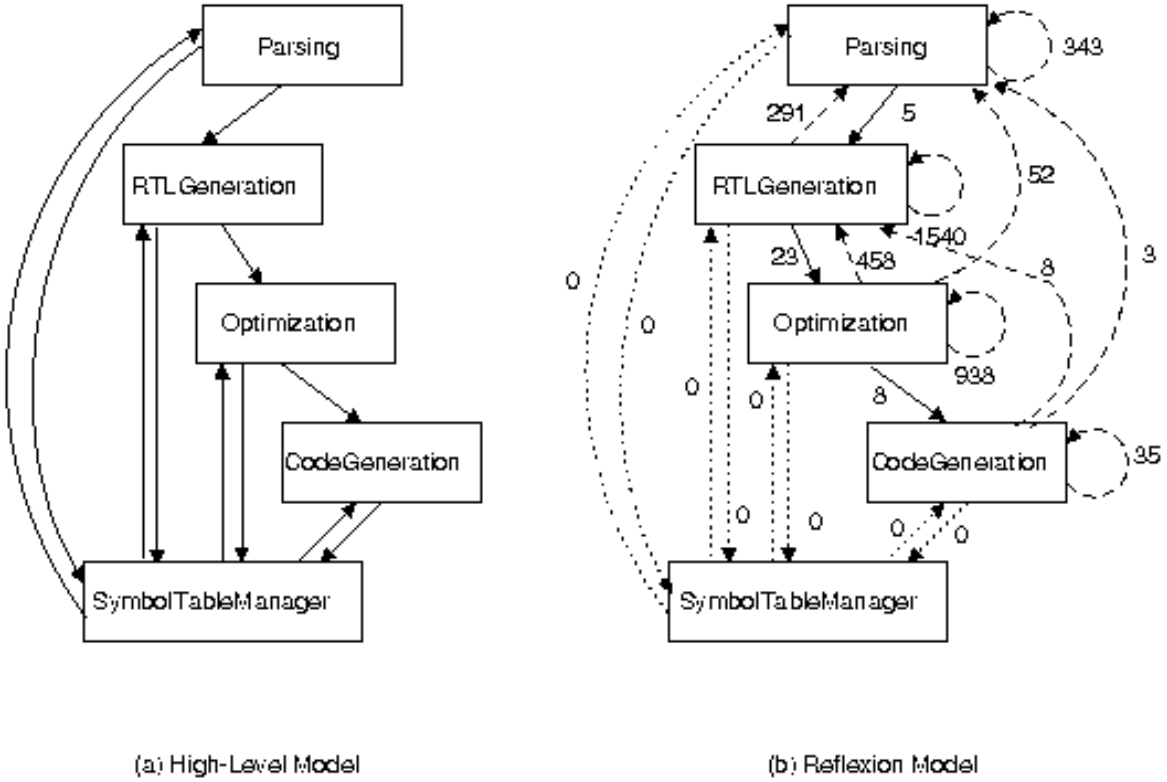
docs.sun.com/source/801-7837/images/6-1-8D82.gif

Beauty or truth?





Maybe a little of both:
Software reflexion models [Murphy 1995]



Lecture schedule and topics:
Subject to change

- October 2: Introduction and Overview
- October 9: Requirements and Specifications I
- October 16: Requirements and Specifications II
- October 23: Design I
- October 30: Design II
- November 6 (likely taught from the Microsoft site): Software Evolution
- November 13: Analysis and Tools
- November 20: Quality assurance/testing
- November 27: Mining software repositories
- December 4: 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
- Details TBD

Mining software repositories:
A relatively new hot research topic

- "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 lecture (December 4)

- TBD
- Jetlagged
- Project presentations?
- Guest speaker?
- Your ideas?

Final examination

- By University rule, an instructor is allowed to dispense with a final examination at the scheduled time (6:30-8:20PM, December 13, 2007) 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 (October 9, 2007)

Assigned work (barring a final):

Four @ 25% each

- Essay on software engineering (1-2 people)
  - Write a 5-10 page essay on what advances in key attributes (productivity, quality, etc.) are within reach by 2010. The essay must not be "pie in the sky" nor primarily personal opinion; rather, it should be scholarly
  - We will post all essays for comments
- Tool-based assignment (individual). Details forthcoming (likely to use Daniel Jackson's Alloy)

Other two assignments

- Two state-of-the-research reports (one in groups of 1-3 people, the other in groups of 2-3 people).
- These are secondary research reports on an approved topic based on significant reading of various pertinent papers and materials (and perhaps some hands-on experience)
- These scholarly reports provide information about the topic and your analysis of it, complete with citations, open questions, etc.
- The list of potential topics is enormous
- We will make these reports available to the entire class and request comments

Note

- In general, the topics covered in lecture will not have associated required readings
- Instead, your in-depth readings will be based on the topics you choose for your two research assignments (and to a lesser degree on the essay assignment)
- I am happy to suggest readings on the lecture material to you

This and that

- Stay tuned to the web page
- Jonathan Beall is our TA
- 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, as needed