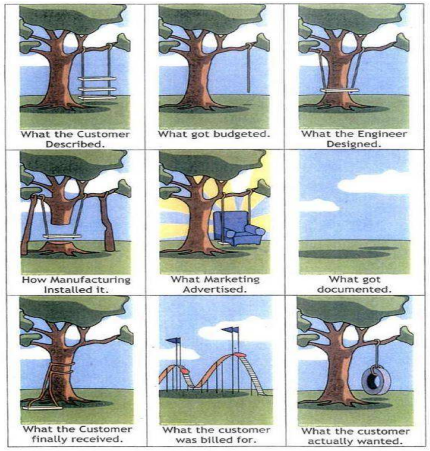
cis20.2
design and implementation of software applications 2
spring 2010
lecture # I.2

today's topics:

- software engineering overview
- software processes

the software world...



software engineering?

- in school, you learn the *mechanics* of programming
- you are given the specifications
- you know that it is possible to write the specified program in the time allotted
- but not so in the real world...
  what if the specifications are not possible? what if the timeframe is not realistic? what if you had to write a program that would last for 10 years?
- in the real world:
  software is usually late, overbudget and broken; and software usually lasts longer than employees or hardware
- the real world is cruel and software is fundamentally brittle
- in addition: *most software ends up being used in very different ways than how it was designed to be used*
- hence the field of **software engineering** was established in order to produce proven methodologies that attempt overcome these real-world issues

software engineering: one definition

- Stephen Schach: "Software engineering is a discipline whose aim is the production of fault-free software, delivered on time and within budget, that satisfies the user's needs."
- software engineering includes the following:
  - requirements analysis
  - human factors
  - functional specification
  - software architecture
  - design methods
  - programming for reliability
  - programming for maintainability
  - team programming methods
  - testing methods
  - configuration management

software engineering: beneficiaries

- the average **manager** has no idea how software needs to be implemented
- the average **customer** says: "build me a system to do X"
- the average **layperson** thinks software can do anything (or nothing)
- the average **programmer** or **software engineer** has the following issues:
  - there is never enough time for development and testing
  - software is often underbudgeted
  - the customer always wants it now! (even though they don't know how long it will take to write it and test it)
  - common pitfalls from customers and management:
    "Why can't you add feature X? It seems so simple..."
    "I thought it would take a week..."
    "We've got to get it out next week. Hire 5 more programmers..."

software engineering: the programmer's dilemma

- you can't do everything yourself
- e.g., your assignment: "write a database-backed web system"
- where do you start?
- what do you need to write?
- do you know how to write a device driver?
- do you know what a device driver is?
- should you integrate a browser into your operating system?
- how do you know if it's working?

software engineering: complexity

- software is complex!
- or it becomes that way
  - feature bloat
  - patching
- e.g., the evolution of Windows NT
  - NT 3.1 had 6,000,000 lines of code
  - NT 3.5 had 9,000,000v
  - NT 4.0 had 16,000,000
  - Windows 2000 has 30-60 million
  - Windows XP has at least 45 million...

software engineering: necessity

- you will need these skills!
- risks of faulty software include
  - loss of money
  - loss of job
  - loss of equipment
  - loss of life
- classic examples
  - therac-25 (`http://sunnyday.mit.edu/papers/therac.pdf`) linear accelerator used in mid-1980's to treat cancer patients with limited doses of radiation; system produced cryptic error messages that operators could not understand and so ignored; patients received overdoses of radiation and some died
  - ariane 501 (`http://sunnyday.mit.edu/papers/jsr.pdf`) european space agency rocket launched in mid-1990's; recalibration routine computed position, velocity, acceleration; one step in recalibration converted floating point value of horizontal velocity to integer, but didn't handle "out of bounds" exception... rocket blew up

software engineering: Fred Brooks

- The Mythical Man-Month (1975)
  - book written after his experiences in the OS/360 design
  - Brooks' Law: "Adding manpower to a late software project makes it later."
  - the "black hole" of large project design: getting stuck and getting out
  - organizing large team projects and communication; documentation!!!
  - when to keep code; when to throw code away
  - dealing with limited machine resources
  - most are supplemented with practical experience
- No Silver Bullet (1986)
  - "There is no single development, in either technology or management technique, which by itself promises even one order-of magnitude improvement within a decade of productivity, in reliability, in simplicity."
  - why? software is inherently complex; many disagree... but no proven counter-argument
  - Brooks' point: there is no *revolution*, but there is *evolution* when it comes to software development

mechanics

- well-established techniques and methodologies:
  - team structures
  - software lifecycle / waterfall model
  - cost and complexity planning / estimation
  - reusability, portability, interoperability, scalability
  - UML, design patterns

team structures

- why Brooks' Law?
  - training time
  - increased communications: pairs grow by n^2 while people/work grows by n
  - how to divide software? this is *not* task sharing
- types of teams
  - democratic
  - "chief programmer"
  - synchronize-and-stabilize teams
  - eXtreme Programming teams
  - pair programming

lifecycles

- software is not a build-one-and-throw-away process
- that's far too expensive
- so software has a *lifecycle*
- we need to implement a *process* so that software is maintained correctly
- examples:
  - build-and-fix
  - waterfall
  - Agile

software lifecycle model

- 7 basic phases (Schach):
  - requirements (2%)
  - specification/analysis (5%)
  - design (6%)
  - implementation (module coding and testing) (12%)
  - integration (8%)
  - maintenance (67%)
  - retirement
- percentages in ()'s are average cost of each task during 1976-1981
- testing and documention should occur throughout each phase
- note which is the most expensive!

requirements phase

- what are we doing, and why?
- need to determine what the client needs, not what the client *wants* or *thinks* they need
- worse — requirements are a moving target!
- common ways of building requirements include:
  - prototyping
  - natural-language requirements document
- use interviews to get information (not easy!)

specification phase

- the "contract" — frequently a legal document
- what the product will do, not how to do it
- should NOT be:
  - ambiguous, e.g., "optimal"
  - incomplete, e.g., omitting modules
  - contradictory
- detailed, to allow cost and duration estimation
- classical vs object-oriented (OO) specification
  - classical: flow chart, data-flow diagram
  - object-oriented: UML

design phase

- the "how" of the project
- fills in the underlying aspects of the specification
- design decisions last a long time!
- even after the finished product
  - maintenance documentation
  - try to leave it open-ended
- architectural design: decompose project into modules
- detailed design: each module (data structures, algorithms)
- UML can also be useful for design

implementation phase

- implement the design in programming language(s)
- observe standardized programming mechanisms
- testing: code review, unit testing
- documentation: commented code, test cases
- integration considerations
  - combine modules and check the whole product
  - top-down vs bottom-up ?
  - testing: product and acceptance testing; code review
  - documentation: commented code, test cases
  - done continually with implementation (can't wait until the last minute!)

maintenance phase

- defined by Schach as *any change*
- by far the most expensive phase
- poor (or lost) documentation often makes the situation even worse
- programmers hate it
- several types:
  - corrective (bugs)
  - perfective (additions to improve)
  - adaptive (system or other underlying changes)
- testing maintenance: regression testing (will it still work now that I've fixed it?)
- documentation: record all the changes made and why, as well as new test cases

retirement phase

- the last phase, of course
- why retire?
  - changes too drastic (e.g., redesign)
  - too many dependencies ("house of cards")
  - no documentation
  - hardware obsolete
- true retirement rate: product no longer useful