cis20.1
design and implementation of software applications I
fall 2007
lecture # IV.1: software engineering

topics:

- software engineering

software engineering: what is it?

- 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."
- includes:
  - 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: why?

- 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 time frame 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
  - software usually lasts longer than employees or hardware
- the real world is cruel and software is fundamentally brittle

software engineering: who?

- 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)
- most software ends up being used in very different ways than how it was designed to be used

software engineering: time

- you never have enough time
- software is often underbudgeted
- the marketing department always wants it tomorrow
- even though they don't know how long it will take to write it and test it
- "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: people

- you can't do everything yourself
- e.g., your assignment: "write an operating 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

examples: therac-25 (1)

- `http://sunnyday.mit.edu/papers/therac.pdf`
- therac-25 was a linear accelerator released in 1982 for cancer treatment by releasing limited doses of radiation
- it was software-controlled as opposed to hardware-controlled (previous versions of the equipment were hardward-controlled)
- it was controlled by a PDP-11; software controlled safety
- in case of error, software was designed to prevent harmful effects

examples: therac-25 (2)

- BUT
- in case of software error, cryptic codes were displayed to the operator, such as: "MALFUNCTION xx"
  where $1 < xx < 64$
- operators became insensitive to these cryptic codes
- they thought it was impossible to overdose a patient
- however, from 1985-1987, six patients received massive overdoses of radiation and several died

examples: therac-25 (3)

- main cause:
- a race condition often happened when operators entered data quickly, then hit the up-arrow key to correct the data and the values were not reset properly
- the manufacturing company never tested quick data entry — their testers weren't that fast since they didn't do data entry on a daily basis
- apparently the problem had existed on earlier models, but a hardware interlock mechanism prevented the software race condition from occurring
- in this version, they took out the hardware interlock mechanism because they trusted the software

examples: ariane 501 (1)

- `http://sunnyday.mit.edu/papers/jsr.pdf`
- next-generation launch vehicle, after ariane 4
- presigious project for ESA
- maiden flight: june 4, 1996
- inertial reference system (IRS), written in ada
  - computed position, velocity, acceleration
  - dual redundancy
  - calibrated on launch pad
  - relibration routine runs after launch (active but not used)
- one step in recalibration converted floating point value of horizontal velocity to integer
- ada automatically throws out of bounds exception if data conversion is out of bounds
- if exception isn't handled... IRS returns diagnostic data instead of position, velocity, acceleration

examples: ariane 501 (2)

- perfect launch
- ariane 501 flies much faster than ariane 4
- horizontal velocity component goes out of bounds
- IRS in both main and redundant systems go into diagnostic mode
- control system receives diagnotic data but interprets it as wierd position data
- attempts to correct it...
- ka-boom!
- failure at altitiude of 2.5 miles
- 25 tons of hydrogen, 130 tons of liquid oxygen, 500 tons of solid propellant

examples: ariane 501 (3)

- expensive failure:
  - ten years
  - $7 billion
- horizontal velocity conversion was deliberately left unchecked
- who is to blame?
- "mistakes were made"
- software had never been tested with actual flight parameters
- problem was easily reproduced in simulation, after the fact

the mythical man-month

- Fred Brooks (1975)
- book written after his experiences in the OS/360 design
- major themes:
  - 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

- paper written in 1986 (Brooks)
- "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
- lots of people disagree(d), but there is no proof of a 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

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

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!)

today's example

- Metro: "I want a kiosk thingy that helps people get between station A and station B."
- what are the requirements?
-
-
-
-
-

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

today's example

- the Metro kiosk
- write a specification to satisfy the requirements
- e.g., all kiosks should reflect trouble with a train
-
-
-
-

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

today's example

- the Metro kiosk
- design one part of the specification
- e.g., how do multiple kiosks send/receive information about trouble with a train?
-
-
-
-
-

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

today's example

- the Metro kiosk
- e.g., how might the system change once it's been implemented?
-
-
-
-
-

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

planning and estimation

- we still need to deal with the bottom line
  - how much will it cost?
  - can you stick to your estimate?
  - how long will it take?
  - can you stick to your estimate?
- how do you measure the product (size, complexity)?
-

reusability

- impediments:
- lack of trust
- logistics of reuse
- loss of knowledge base
- mismatch of features

reusability: how to

- libraries
- APIs
- system calls
- objects (OOP)
- frameworks (a generic body into which you add your particular code)

portability

- Java and C#
- Java: uses a JVM
  - write once, run anywhere (sorta, kinda)
- C#: also uses a JVM
  - emphasizes mobile *data* rather than code
- winner?
  - betting against Microsoft is historically a losing proposition...

interoperabilty

- e.g., CORBA
- define abstract services
- allow programs in any language to access services in any language in any location
- object-ish

scalability

- something to keep in mind
- don't worry about scaling beyond the abilities of the machine
- avoid unnecessary barriers
- from single connection to forking processes to threads...