cis20.2
design and implementation of software applications 2
spring 2010
lecture # I.4

today's topics:

- software testing

software testing

- "the process of executing a program with the intent of finding errors" [Myers 1979]
- physical systems tend to fail in a few, small (fixed) set of ways; software systems tend to fail in many (strange) ways
- physical systems tend to fail due to manufacturer errors; software systems tend to fail due to design errors
- physical systems tend to fail with age, usage; software systems can fail at any time...
- even small modules can be computationally complex
- exhaustive testing is not tractable: a program that adds two 32-bit integers would take hundreds of years to test exhaustively (2^{64} distinct test cases)
- fixing bugs may introduce new (often more subtle) bugs

why test software?

- to improve quality
  - bugs can be costly ($ and lives... remember examples of ariane and therac)
  - quality implies conforming to design requirements
- for verification and validation
  - funcionality (exterior quality)
  - engineering (interior quality)
  - adaptability (future quality)
- for reliability estimation

classifications

- by purpose:
  - correctness testing
  - performance testing
  - reliability testing
  - security testing
- by life-cycle phase:
  - requirements phase testing
  - design phase testing
  - program phase testing
  - evaluation test results
  - installation phase testing
  - acceptance testing
  - maintenance testing

- by scope
  - unit testing
  - component testing
  - integration testing
  - system testing

correctness testing

- minimum requirement of software testing
- need to be able to tell correct from incorrect behavior
- "white-box" and "black-box" methods
- black-box testing
  - also called "data driven" testing
  - test data are derived from functional requirements
  - testing involves providing inputs to a module and testing the resulting outputs; hence the name "black box"
  - only testing the functionality
- white-box testing
  - also called "glass box" testing
  - structure and flow of module being tested is visible
  - test cases are derived from program structure
  - some degree of exhaustion is desirable, e.g., executing every line of code at least once

- other methods: control flow testing, mutation testing, random testing
- control flow testing
  - also called/includes loop testing and data-flow testing
  - program flow is mapped in a flowchart
  - code is tested according to this flow
  - can be used to eliminate redundant or unused code
- mutation testing
  - original program code is perturbed and result is many new programs
  - all are tested—the most effective test cases/data are chosen based on which eliminate the most mutant programs
  - but this is (even more) exhaustive and intractable
- random testing
  - test caes are chosen randomly
  - cost effective, but won't necessarily hit all the important cases
- combinations of above yield best results in terms of completeness/effectiveness of testing, tractability and cost

performance testing

- e.g., make sure that software doesn't take infinite time to run or require infinite resources
- performance evaluation considers:
  - resource usage
    e.g., network bandwidth requirements, CPU cycles, disk space, disk access operations, memory usage
  - throughput
  - stimulus-response timing
  - queue lengths
- benchmarking frequently used here

reliability testing

- determination of failure-free system operation
- dependable software does not fail in unexpected or catastrophic ways
- "robustness" means the degree to which software can function when it receives unexpected inputs or within stressful conditions
- "stress testing" or "load testing" pushes the software/system beyond the typical limits to see if/when it will fail

security testing

- refers to testing for flaws that can be exploited by security holes
- particularly relevant to software that runs on the internet
- simulated security attacks help test this category

testing automation

- software testing tools help cut costs of manual testing
- typically involve the use of test scripts
- which are also costly to create
- so are used in cases where they are less costly to create and run than manual testing

when to stop?

- never! (hehe)
- there are always two more bugs—the one you know about and the one you don't...
- trade-offs between budget, time, quality
- choices must be made...
- alternatives?
  - buggy software/systems?
  - some kind(s) of testing is necessary
  - "proofs" using formal methods
  - do you think the use of design patterns may help reduce testing?

software testing: conclusions

- software testing is an art
- testing is more than just debugging
- testing is expensive
- complete testing is infeasible
- testing may not be the most effective way to improve software quality

test scenarios

- a *scenario* is a "hypothetical story used to help a person think through a complex problem or system"
- the idea of "scenario planning" gained popularity in the 2nd half of the 1900's
- can be useful for illustrating a point, as well as testing a system
- ideal scenario has five characteristics:
  - a story that is
  - motivating
  - credible
  - complex
  - easy to evaluate
- storytelling is an art, but a good scenario is a good story!
- risks and problems:
  - other approaches are better for testing early code
  - not designed for coverage of an entire system

  - often heavily documented and used repeatedly, but often expose design errors rather than implementation (coding) errors
- scenario testing vs requirements analysis
  - requirements analysis is used to create agreement about a system to be built; scenario testing is used to predict problems with a system
  - scenario testing only needs to point out problems, not solve them, reach conclusions or make recommendations about what to do about them
  - scenario testing does not make design trade-offs, but can expose consequences of trade-offs
  - scenario testing does not need to be exhaustive, only useful

creating test scenarios

- write life histories for objects in the system
- list possible users; analyze their interests and objectives
- list "system events" and "special events"
- list benefits and create end-to-end tasks to check them
- interview users about famous challenges and failures of the old system
- work alongside users to see how they work and what they do
- read aobut what systems like this are supposed to do
- study complaints about he predecessor to this system and/or its competitors
- create a mock business; treat it as real and process its data

(from Kaner article)