Lecture Notes
September 27, 2018

Team assignment Preliminary Project Proposal due tonight, 11:59pm.

The teaching staff will review the preliminary project proposals, and then a specific IA mentor will be assigned to each team. The Revised Project Proposal is due Tuesday, October 9, 11:59pm.

The entire team should meet with the team's IA mentor to discuss the team project as soon as possible, and in any case *before* submitting the revision (CVN teams should arrange for an online meeting with the designated CVN IA = Anthony). The revised proposal should take into consideration feedback provided by your IA mentor and any other members of the teaching staff who make comments.

Continuing testing overview: Ideally, testing is *automated* to remove human tedium and error and make testing reproducible, faster, more likely to be done

The simplest automated testing is provided by unit testing tools. A unit is a method (or function, procedure, subroutine) or a class (or module, file)

Unit testing tools *execute* all the test cases you provide for each unit, and report the results.

They do not *write* the test cases for you - although some may generate test skeletons or templates to fill in. (Some research-quality tools *do* generate test cases, e.g., Evosuite and Randoop).

Popular unit testing tools: JUnit, TestNG, xUnit

A unit testing "runner" works when the software under test can be invoked from other code. Thus unit testing tools can run integration tests too.

Specialized testing frameworks tools support invoking the code from the UI (e.g., Selenium automates web browser applications and Appium automates mobile apps)

A unit test [is]

- Identified by naming convention or annotation
- Usually maintained in a folder hierarchy parallel to the folder hierarchy of corresponding source code
- Constructs the application state and parameters for method under test
- Invokes that method with the input parameters in the context of the application state
- Supplies non-parameter inputs, e.g., inputs from (usually simulated) human user, file, network, database, device, etc.
- Collects the results of that method, including both output parameters and changes to application state
- Applies a set of assertions to check that the results are as expected
- There may be a special notation for writing assertions or this may be regular code (which could itself have bugs)

Unit tests for a unit should be written as soon as the unit has been written (or even before starting to write the unit = [test-driven development])

Automated testing tools provide way to associate setup and teardown code with each test case (or with test class = a group of test cases), called a "fixture". Fixtures are necessary to ensure tests are *repeatable*

Setup invoked before a single unit test, or at the beginning of a test class with multiple unit tests, to set up application state that simulates the state the program is expected to be in when the method(s) would normally be invoked with the given parameters during full system execution

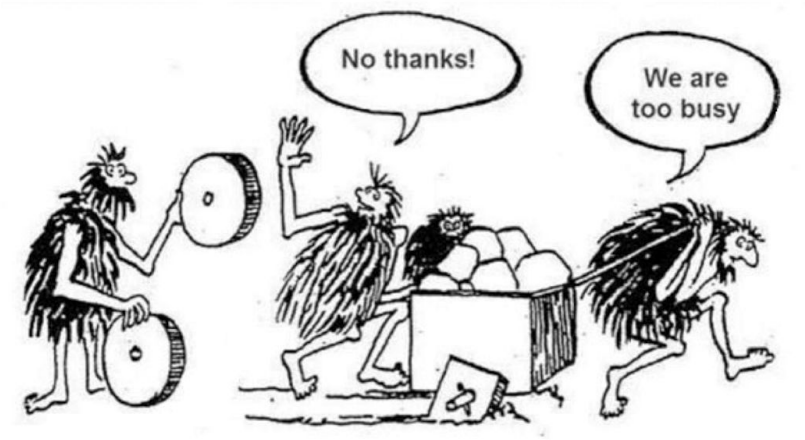
When there are multiple different states where the same method or set of methods might be called, then there should be multiple different setups - each with its own corresponding teardown

Teardown code should restore memory, file system, network, database, etc. to "clean" state, with no traces of the tests that just completed so setup for the next tests starts in pristine state

Automated testing often uses mocks rather than real resources (e.g., EasyMock and Mockito, Powermock)

- ➢Not always feasible or cost-effective to put real-world resources like enterprise database into desired pre-test state and cleanup post-test

Top 10 Excuses Programmers Give to Avoid Unit Testing



- ➢Demo JUnit (Sara and Kiran)

- ➢Demo Jest (Fujunku)

Most modern version control systems and automated testing tools work together with compilers, package managers, etc. to support *continuous integration*, a key feature of agile processes

Every time a changed file or set of files is committed to the shared repository, the entire codebase is built and tested so errors are detected immediately

- External "build" tools hooked to the VCS are automatically invoked before (*pre-commit* on local machine) and after (*post-commit* on server) every commit to the shared repository
- Does whatever is needed to make the system executable, so developers do not need to remember detailed command options
- Then runs incremental test suite locally before commit and full test suite on server immediately after commit (or nightly)
- Possibly also runs style checkers, static analysis bug finders, and other tools
- Sometimes *continuous delivery* - automated deployment - on demand or at preset times, to install within company (devops) or ship (customer)

More continuous integration on Tuesday...

How do you know when you're "done" testing?

- You're *never* done if you keep changing the code (regression testing).
- This is why many software organizations "freeze" a planned release build a week or more in advance, with no changes allowed except bug fixes (with each set of repairs followed by full re-testing)

Dijkstra: "Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."

How do you know when you have "enough" tests?

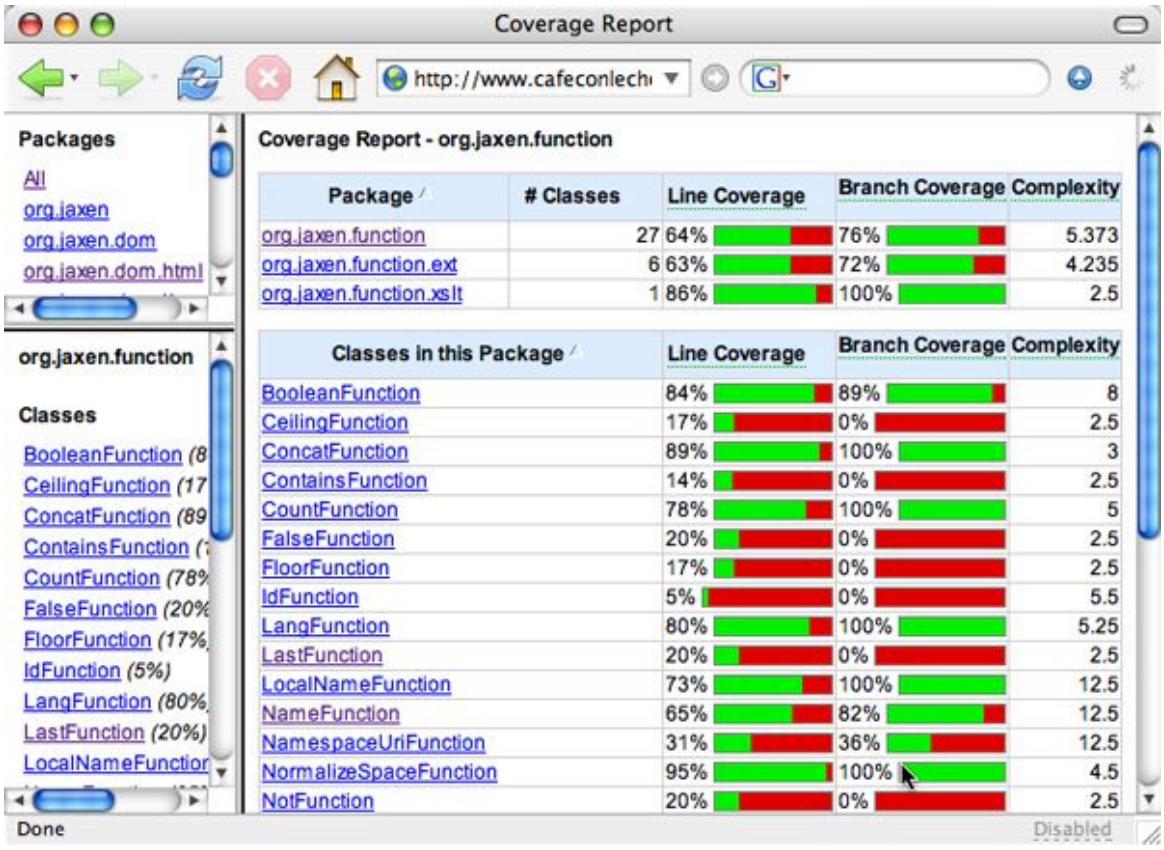
- You don't. Infeasible to test exhaustively, but you need to stop somewhere.
- Several techniques for deciding whether a test suite is sufficient and determining what is missing.

Coverage (aka "test adequacy"): Exercise [close to] 100% of statements, branches, conditions (truth tables), control flow paths, data flow paths (def-use), state transitions, grammar, etc.

Part 2 of textbook goes into great detail about many different kinds of coverage, but we will only consider basic statement, branch and condition coverage

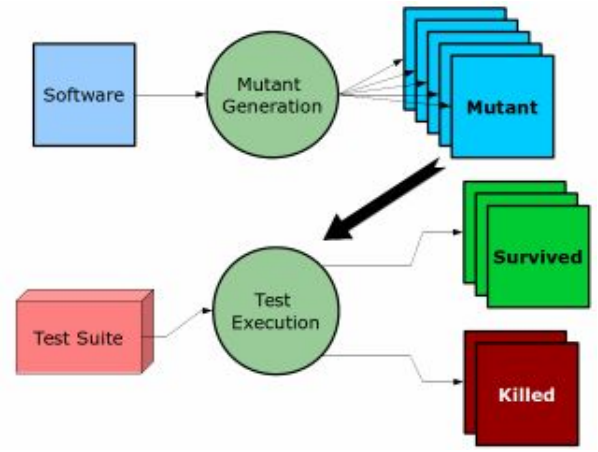
Intuition: If you have never run that part of the code, how can you have any confidence that it works?

Coverage testing tools instrument the test cases and/or the software under test to track coverage metrics across *all* test cases executed, inform what has not yet been covered



Coverage checking adds overhead to testing, but not to production

Mutation analysis: Mutation testing tools automatically generate many program variants, called "mutants", each with one mutation



First order mutant:

- make small change in an individual statement or condition, e.g., change > to <,
- change value of a constant,
- change the use of a variable to use a different variable with the same type,
- delete a statement,
- insert a return statement,
- change synchronization variable,
- add synchronization statement at arbitrary place

Second order mutant: make coordinated changes in two or more different places

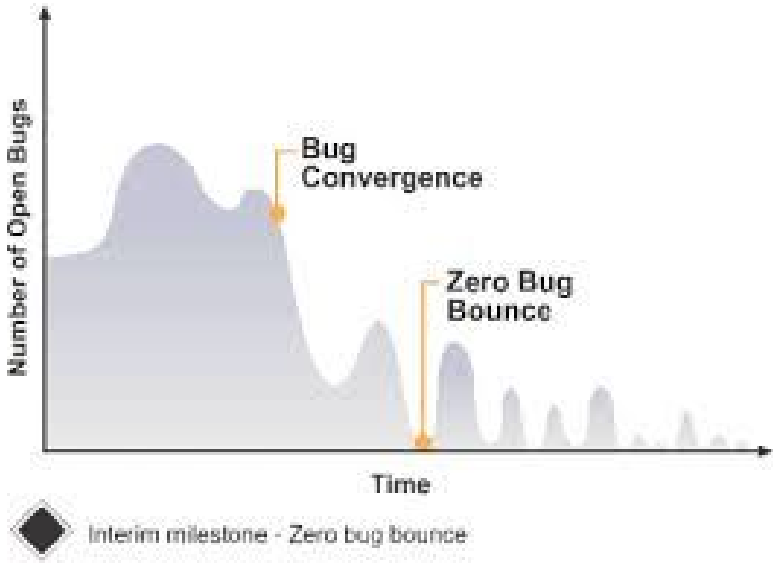
Mutations proxy for (generic) real bugs that happen frequently, except you know where they are

Intuition: If the test suite does not “kill” all the mutants, how can you expect it to find all real bugs?

Mutation testing can be enormously expensive - why?

Zero bug bounce: Rate of finding new bugs is stable and low (approaching zero)

Intuition: Reached point of diminishing returns



6 STAGES OF DEBUGGING

1. That can't happen.
2. That doesn't happen on my machine.
3. That shouldn't happen.
4. Why does that happen?
5. Oh, I see.
6. How did that ever work?

Read for Tuesday:
textbook, chapter 2 Model-Driven Test Design

Pair assignment Practice with Bug Finders due before class on Tuesday

Team assignment Revised Project Proposal due Tuesday, October 9, 11:59pm

You will be informed who is your IA mentor via a comment on your Preliminary Proposal in canvas - arrange to meet your IA mentor asap to discuss your proposal and appropriate revisions