Lecture Notes
October 24, 2017

Implementation team assignment due Thursday (schedule demo with your TA mentor)

Midterm 360-degree Feedback due Friday

Black-box unit testing and completion of first iteration due November 9 (another demo)

Today's lecture (and readings) and thereafter subject to second exam at end of semester

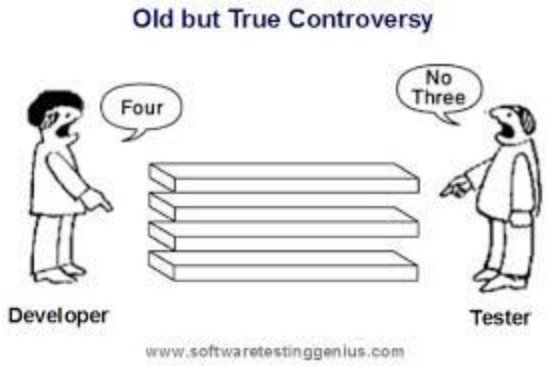
Testing Overview

Many bugs can be found by *static* analyzers, which examine the source code to find “generic” bugs and suspicious code patterns that appear in many different kinds of programs

Static analysis on a program often produces *false positives*, reporting bugs that could never happen during actual execution because it considers all paths through the program - some of which may not be feasible at runtime, why not?

Static analysis can also be applied to bytecode or binaries, not necessarily to find bugs - often used to find malware

To find application-specific or domain-specific bugs in your program, you need to test the features - *dynamic* because the program is executed

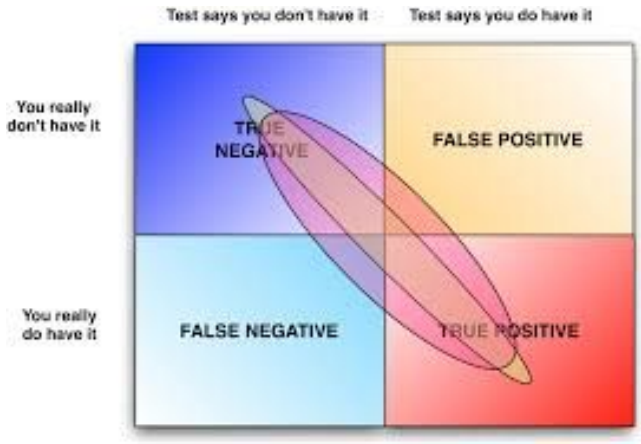


Bugs found during full program testing are never false positives, because not only *could* they happen during actual execution, they *did* happen

There are “generic” dynamic analyzers (e.g., fuzzing), which don’t know what your code is supposed to do but can find some bugs - particularly crashing bugs

Fuzzing takes existing test cases and “fuzzes” the inputs with semi-random changes

Both static and dynamic analysis exhibit *false negatives*, failing to report all bugs - there is no technique that can be guaranteed to find *all* bugs in a non-trivial program written in a conventional programming language



Testing Vocabulary:

Functional testing: Unit testing, integration testing, system testing, acceptance testing

Unit testing - tests individual methods, classes or other “units”, typically in isolation from the rest of the program (using mocks/stubs to substitute for rest of program)

Integration testing - tests boundaries between “units” (replace mocks/stubs with real program units one at a time), term also refers to testing interfaces to external libraries, systems, databases, etc.

System testing - aka end-to-end testing, tests full system from external user-visible entry points

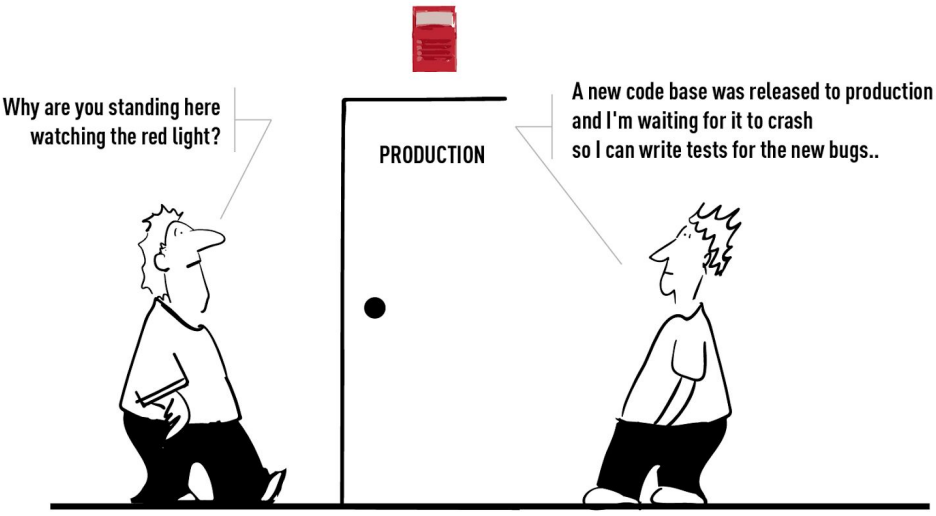
User Acceptance Testing (UAT) - system testing in customer’s environment with real-world users and (sometimes) real-world workloads, beta testing

Non-functional testing: compatibility testing, performance testing, penetration and other security testing, fault injection, UI/UX testing, etc.

Test-to-pass: initial testing makes sure the software minimally works (“smoke test”)

Test-to-fail: most testing is trying to find bugs, so they can be fixed before deployment

> Much more expensive to fix a bug found by a customer/user than to fix it before the customer/user gets a chance to encounter it

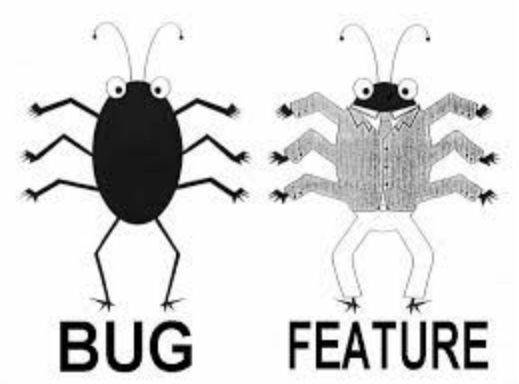


What is a bug?

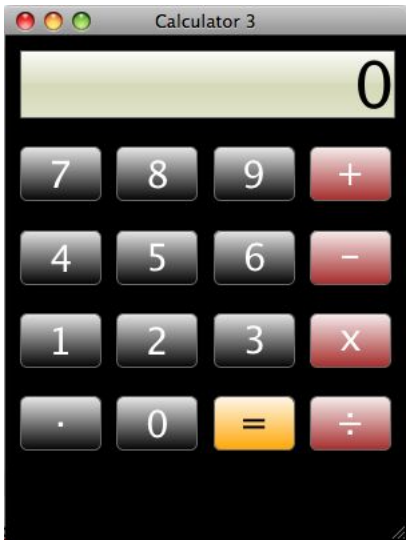
Bugs are not always in code, could be in the requirements, design, tests or documentation

Broad concept of what is a bug:

- Software doesn't do something requirements say it should do
- Software does something requirements say it shouldn't do
- Software does something that requirements don't mention
- Software doesn't do something that requirements don't mention but should
- Software is difficult to understand, hard to use, slow, etc. - users will consider this a bug



Consider a simple calculator:



The tester enters a number, presses +, enters another number, presses =

- Nothing happens
- Computes the wrong answer

The calculator works correctly for some period of time or some number of computations, and then stops responding

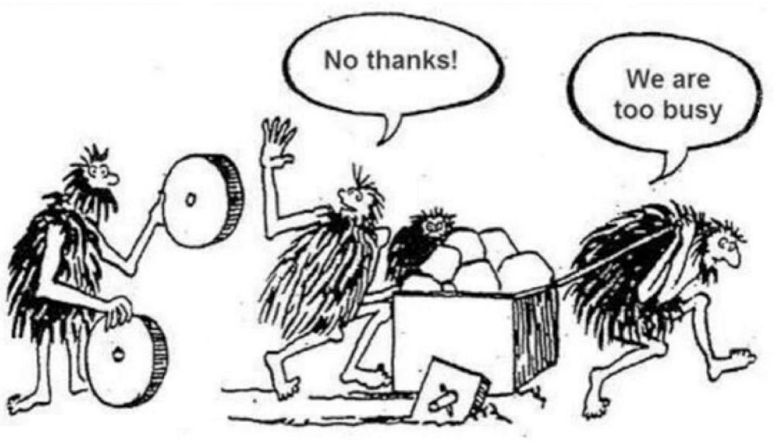
The calculator displays all 0's and won't do anything else

The calculator correctly adds, subtracts, multiplies and divides, but the tester found that if two operators are held down simultaneously, it appears to do square root

The calculator's buttons are too small, the = key is in an odd place, the display cannot be read in bright light, ...

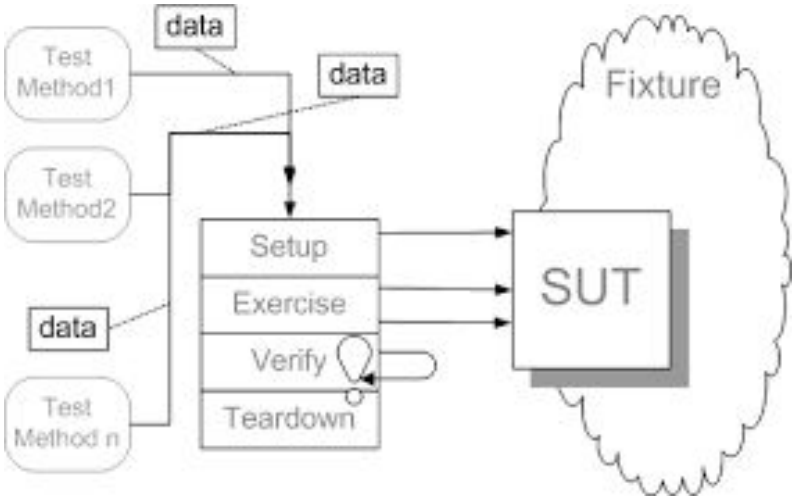
Ideally, testing is *automated*

- Why automate? Removes human tedium and error, reproducible, faster
- Testing framework provides “runner” that invokes series of test cases (e.g., unittest)
- Specialized frameworks for UI testing (e.g., selenium for web applications)



Associate setup/teardown code with every test case (or with every group of test cases), sometimes called “fixture”

> *Setup* code usually assumes a “clean” application state, and puts the system into the state necessary to conduct the test
>
> *Teardown* code should restore memory, file system, network, database, etc. to “clean”, state with no traces of the test that just completed



Testers should avoid/prevent dependencies among tests, but some test runners restart/reboot the environment between tests just in case

> May be very time-consuming - might take 10ms to run each test and 2s to restart JVM - much longer to reboot Linux and most other OS

Need to be able to run tests in any order with same results, no “flaky tests” (appear to randomly pass or fail)

Testers often use mocks and stubs rather than real resources

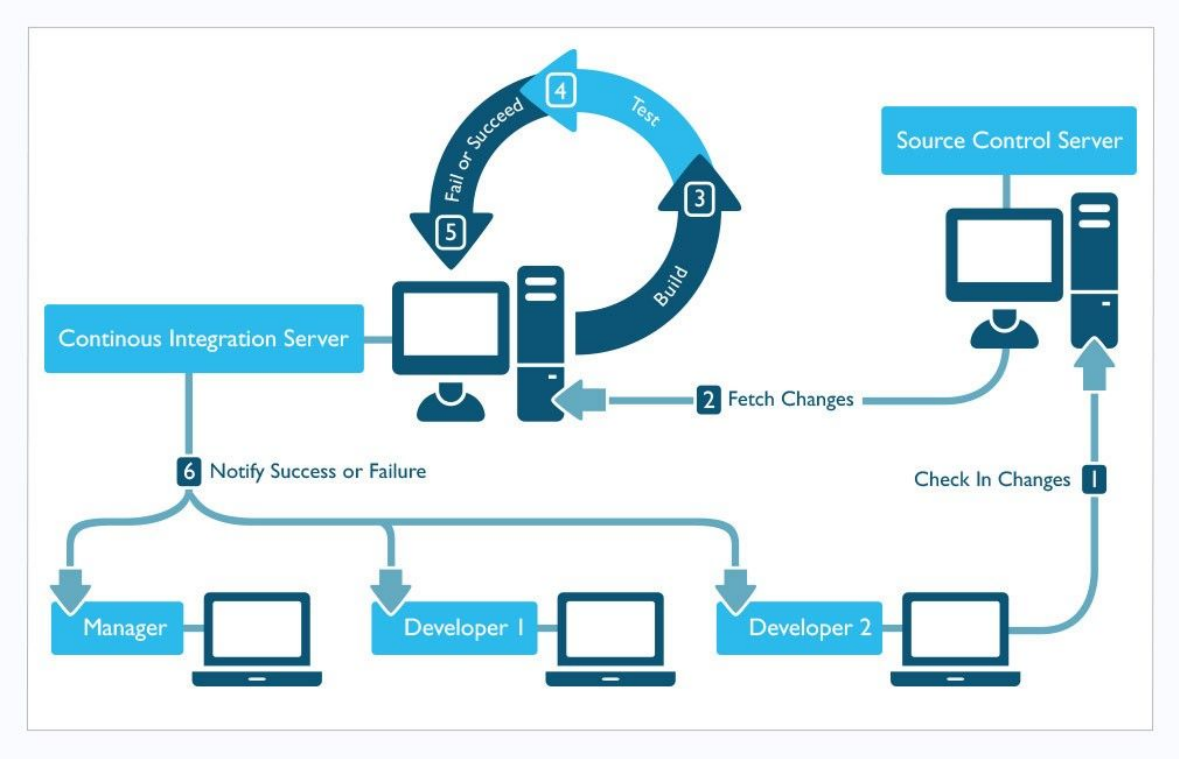
> Not always feasible to put real-world resources into desired pre-test state and cleanup post-test because of visible external side-effects
>
> Even when no external sharing, may take too long

Continuous integration = “build” includes running full regression test suite

> “Regression” is when new/modified code causes previously passing test cases to fail

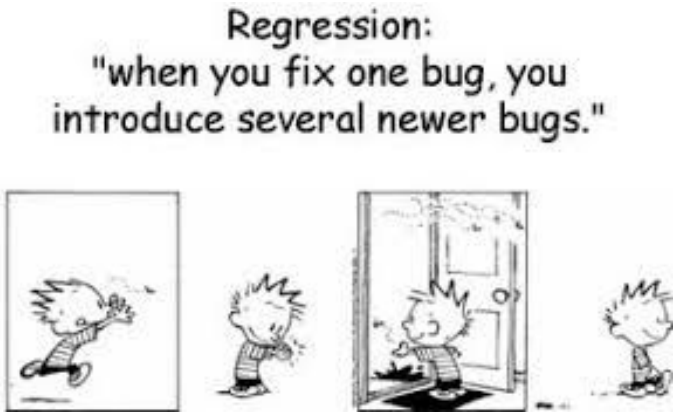
On every commit, on demand, or on a fixed schedule such as nightly

> The longer the test suite takes to run, the less often it will be run - there are tools to help prioritize, reduce, select within or “minimize” test suites



Continuous deployment = When passed build also deploys (devops or patch)

How do you know when you're "done" testing? You're never done if you keep changing the code, need to re-run test suite after every change (regression testing)

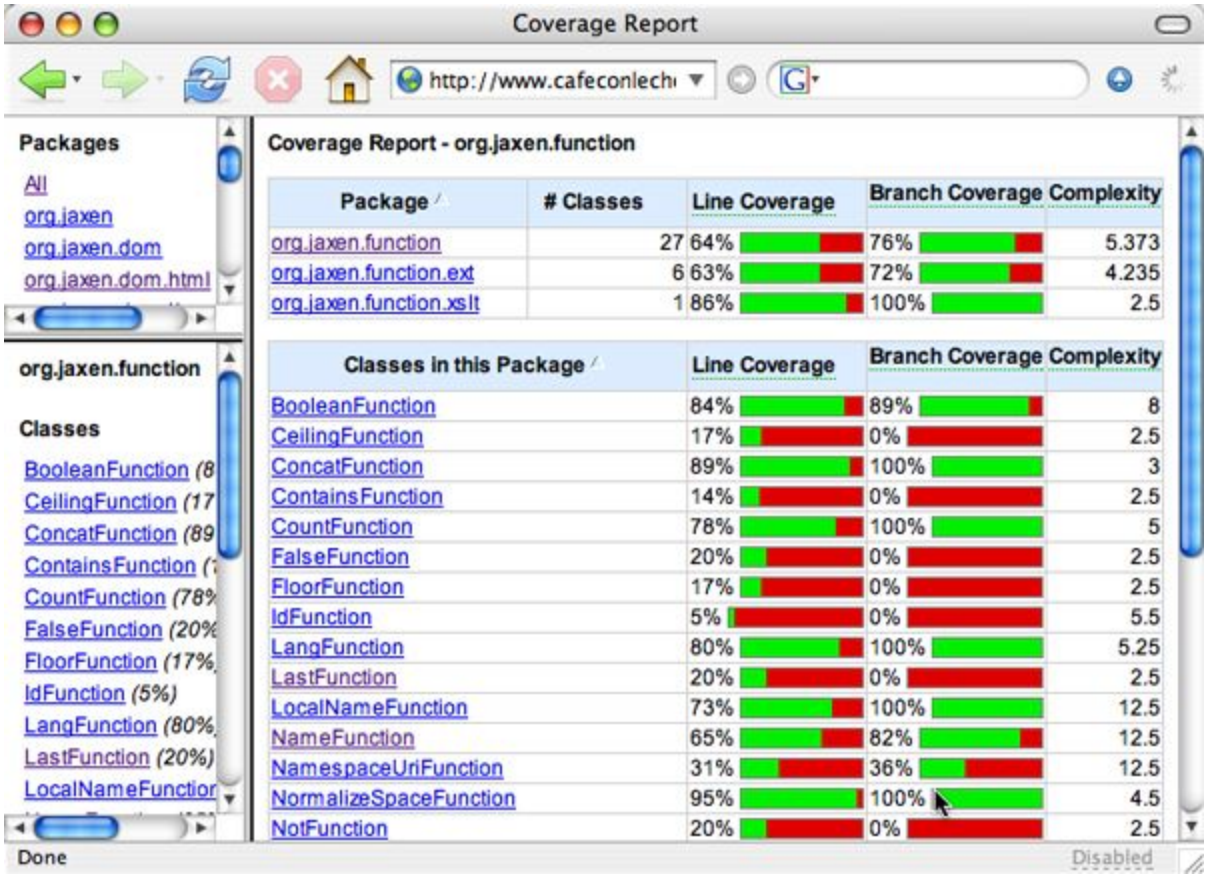


How do you know when you have "enough" tests in the test suites? You don't, but you need to stop somewhere, use some criteria

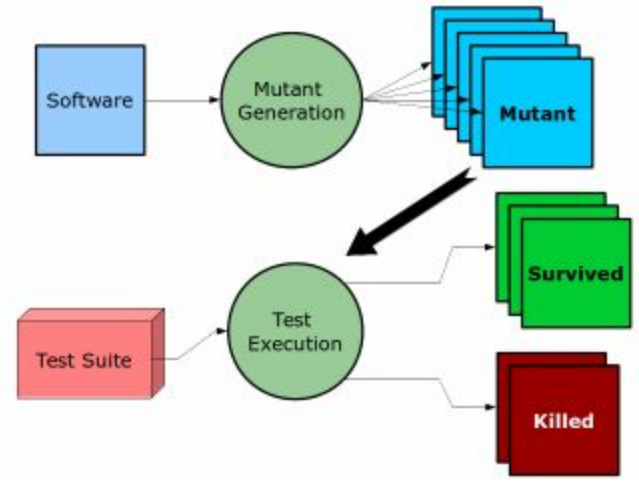
Dijkstra: "Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence." Cannot test exhaustively.

Coverage (aka adequacy): Exercise [close to] 100% of statements, branches, branch conditions (truth tables), control flow paths, data flow paths (def-use)

If you have never exercised that code, how can you have any confidence that it works?



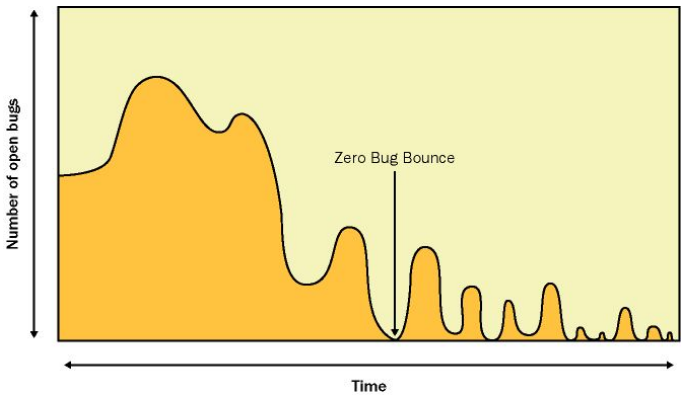
Mutation analysis: Automatically generate program variants, called “mutants”, each with one mutation - Mutations proxy for real bugs, but you know where they are. Does you test suite find all the mutants? If not, how can you expect it to find the real bugs?



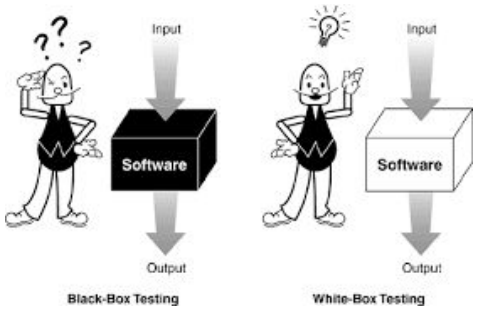
- First order mutant: make small change in an individual statement or condition, e.g., change > to <, change value of a constant, change the use of a variable to use a different variable with the same type, delete a statement, insert a return statement, change synchronization variable, add synchronization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

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

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



Functional tests divided into black box vs. gray box vs. white box testing



Black box - tester considers what the box is supposed to do, not how it does it, doesn't look inside the box

> The "box" is different at different levels, e.g., for unit testing the box is the unit, for system testing its the full application
>
> May miss corner cases

Users see the system from the outside, closed black box, only concerned with functionality

> Invalid input should be rejected in a way that a typical user can understand
>
> Error messages (or lack thereof) are typically the last thing developers think about but the first thing users notice when the software does not work the way users expected
>
> Software that doesn't work from user perspective won't get used

Gray box - tester looks at intermediate products between boxes and leftover side-effects after box is done executing (e.g., network I/O, log entries)

> Testers peek under the covers a little, checking data in the database, opened network connections are closed, memory usage is steady
>
> Particularly important for web/mobile and other applications where a client interacts with a server that resides on the network and data is stored in a shared database

White box - tester leverages full knowledge of what is inside the box

> Necessary to achieve coverage - check what hasn't been covered by other tests and construct test inputs specifically to force execution of previously unexercised code
>
> Cannot detect *missing* functionality

Stopped here

Unit Testing

The simple testing done by novice programmers, if they test at all, is typically system testing - the program is run from the command line or GUI with a few sample inputs

> These are usually test-to-pass

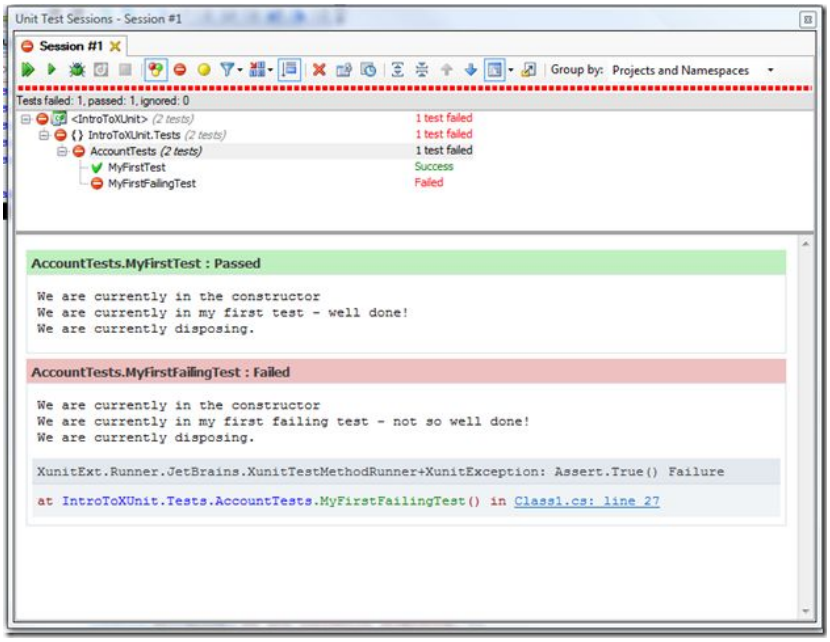
Focus of professional testing is test-to-fail, at all levels

> If units (that are actually used by the full system) misbehave, it is unlikely that the full system behaves properly even if it appears to do so from a few system tests
>
> The initial unit tests are usually written by the developers who wrote or changed the unit, as part of completing the user story, but more extensive unit tests may be written by separate testers (e.g., to achieve coverage)

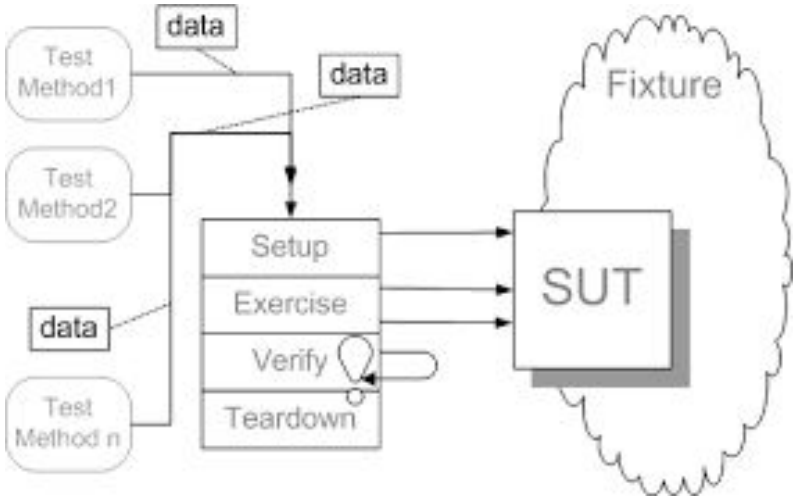
What is a “unit”? When I say unit, I’m usually thinking of a method, function, procedure, subroutine, but it is not unusual for software engineers to treat a class, module, or other program component as a unit for unit testing purposes

> There are almost always *multiple* test cases for each unit, not just one

Unit testing tools, often named “xUnit” for some language-specific x, enable running a set of unit tests on demand



Usually four phases for each test case: setup, exercise, verify, teardown



Exercise means actually running the test subject (“code under test”, “method under test”)

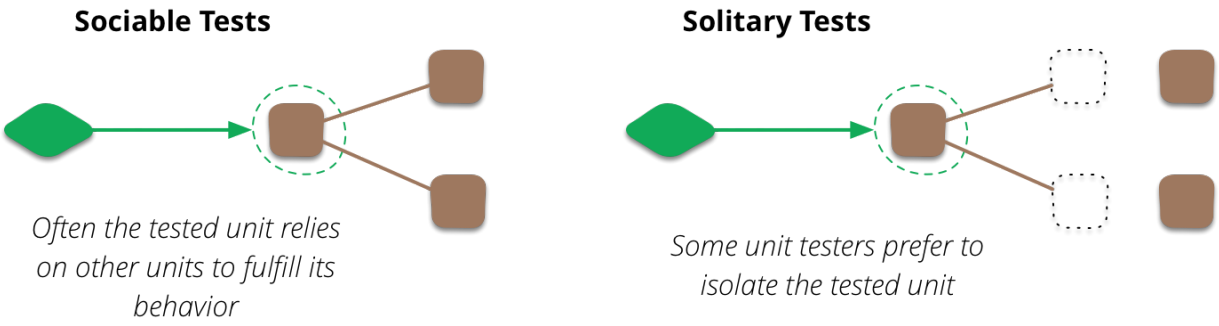
Verify means checking that the actual result matches the expected result (need a “test oracle” to know what is the expected result - for the kinds of applications considered in this class, the test oracle is usually the human tester)

The “result” is not necessarily a return value, may be a change to state that persists beyond the unit under test - State verification

(see Mocks Aren’t Stubs for discussion of mock-based Behavior verification)

Mocks and Stubs

"Sociable" tests vs. "Solitary" tests



For the purposes of this course, unit tests are solitary tests

Fowler's explanation of sociable tests (in https://martinfowler.com/bliki/UnitTest.html, assigned reading) is incomplete/misleading, because to truly test a unit in the context of the whole program, you need the code/context "above" that unit not just the code/context "below"

> I refer to full-context unit tests as "in-vivo" tests, vs. solitary tests as "in-vitro" tests, but you do not need to be concerned with in-vivo testing for this course

How do you *isolate* the tested unit, for solitary testing, if it calls other methods and/or reads state besides local variables and parameters and/or changes the values of parameters and other non-local state that persists after the method returns?

> And if you do manage to isolate the unit under test, how do you verify state changes to state outside that unit?

Fowler refers to "TestDoubles", which includes full range of fake non-local objects and methods that might be used during isolated unit testing

TestDoubles are needed for two main purposes:

1. Make the compiler/runtime happy, because the referenced objects and methods need to exist even if they are never used by a given test
2. Standing in for the relevant parts of the rest of the program (or external resources) that indeed are actually used during one or more unit tests, so need to be/do <something> useful for the given unit test

First consider “never used” case:

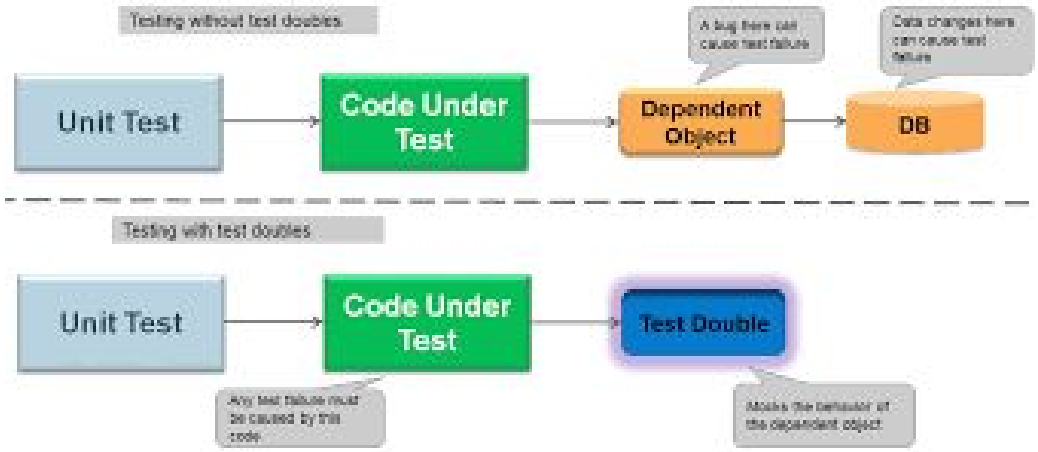
Any other methods that can be called anywhere during the method under test have to exist, even if never actually called during the given test case

> fakeMethod (same parameter list as real method) : same return type as real method
> { return fakeObject of correct return type }

Need to call the method under test with all parameters required by the compiler/runtime, so need to conjure up fake parameters during setup before the actual unit test begins and then get rid of these fake parameters during teardown after the unit test ends

If any part of a method accesses non-local state, even if that part is not actually executed during the given unit test, still usually need setup/teardown for this state

Now consider “actually used” case:



Stubs hardcode a specific state, operation and/or return value

Mocks can be configured to produce varying states and behaviors for testing purposes

Example with stubs:

We want to take an order object and fill it from a warehouse object. The order is very simple, with only one product and a quantity. The warehouse holds inventories of different products. When we ask an order to fill itself from a warehouse there are two possible responses. If there's enough product in the warehouse to fill the order, the order becomes filled and the warehouse's amount of the product is reduced by the appropriate amount. If there isn't enough product in the warehouse then the order isn't filled and nothing happens in the warehouse.

```
public class OrderStateTester extends TestCase {
  private static String TALISKER = "Talisker";
  private static String HIGHLAND_PARK = "Highland Park";
  private Warehouse warehouse = new WarehouseImpl();

  protected void setUp() throws Exception {
    warehouse.add(TALISKER, 50);
    warehouse.add(HIGHLAND_PARK, 25);
  }
  public void testOrderIsFilledIfEnoughInWarehouse() {
    Order order = new Order(TALISKER, 50);
    order.fill(warehouse);
    assertTrue(order.isFilled());
    assertEquals(0, warehouse.getInventory(TALISKER));
  }
  public void testOrderDoesNotRemoveIfNotEnough() {
    Order order = new Order(TALISKER, 51);
    order.fill(warehouse);
    assertFalse(order.isFilled());
    assertEquals(50, warehouse.getInventory(TALISKER));
  }
```

Do you see any problems with these tests?

Same example with mocks (using record/replay, there are other approaches to configuring):

```
public class OrderEasyTester extends TestCase {
  private static String TALISKER = "Talisker";

  private MockControl warehouseControl;
  private Warehouse warehouseMock;

  public void setUp() {
    warehouseControl = MockControl.createControl(Warehouse.class);
    warehouseMock = (Warehouse) warehouseControl.getMock();
  }

  public void testFillingRemovesInventoryIfInStock() {
    //setup - data
    Order order = new Order(TALISKER, 50);

    //setup - expectations
    warehouseMock.hasInventory(TALISKER, 50);
    warehouseControl.setReturnValue(true);
    warehouseMock.remove(TALISKER, 50);
    warehouseControl.replay();

    //exercise
    order.fill(warehouseMock);

    //verify
    warehouseControl.verify();
    assertTrue(order.isFilled());
  }

  public void testFillingDoesNotRemoveIfNotEnoughInStock() {
    Order order = new Order(TALISKER, 51);

    warehouseMock.hasInventory(TALISKER, 51);
    warehouseControl.setReturnValue(false);
    warehouseControl.replay();

    order.fill((Warehouse) warehouseMock);

    assertFalse(order.isFilled());
    warehouseControl.verify();
  }
}
```

It is ok to use only stubs, not mocks, for this course - but if you would like to use mocks, find out which mock tool(s) play nicely with your team’s choice of implementation stack