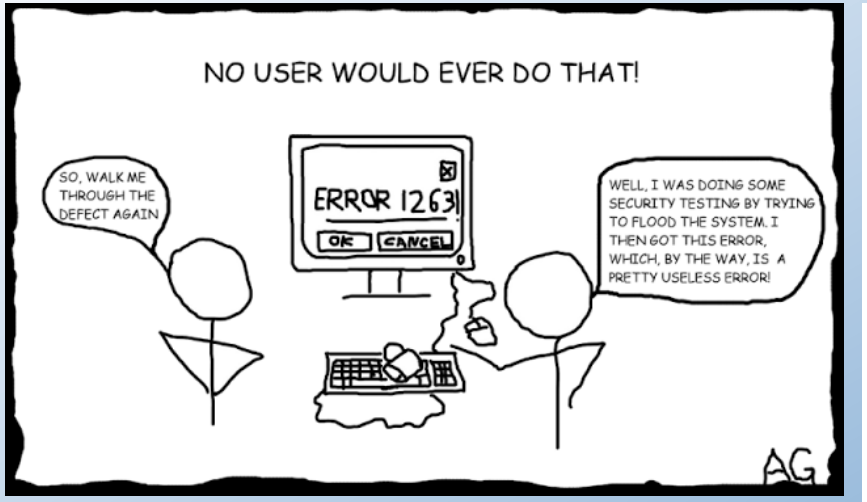
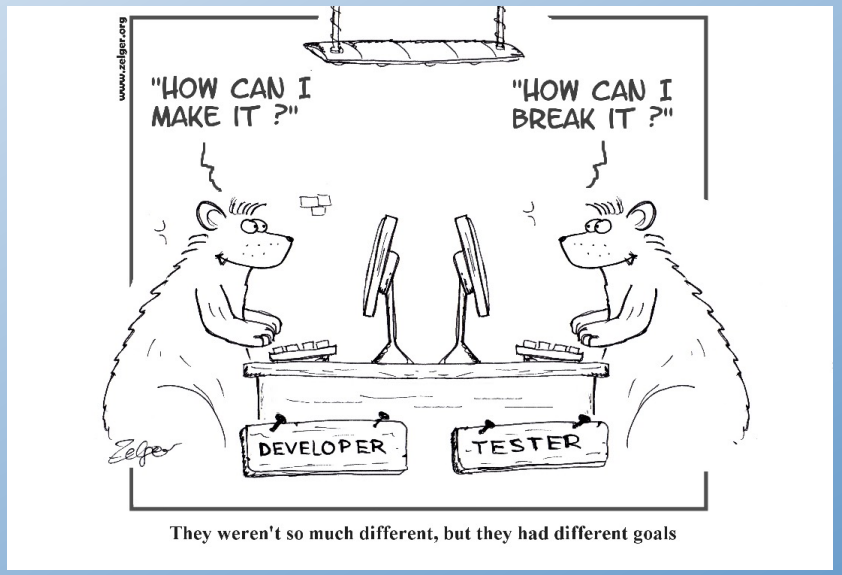
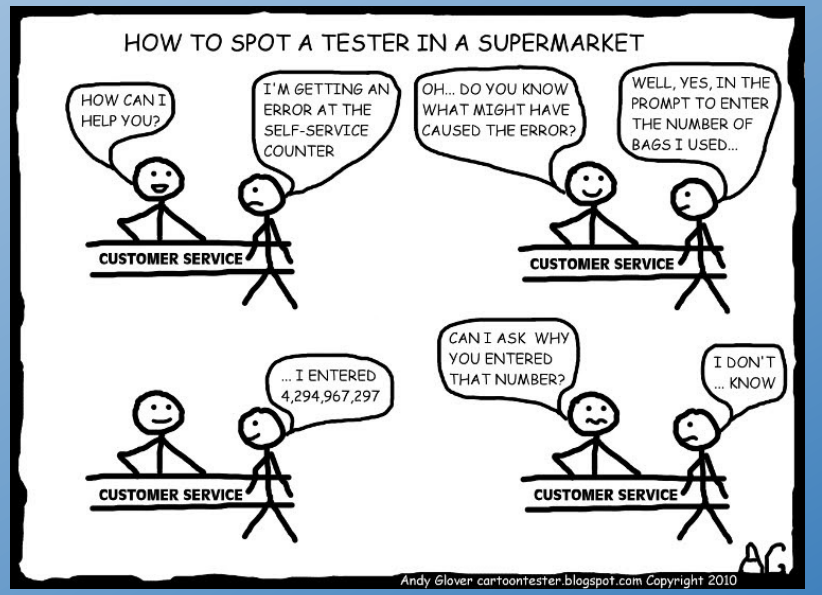


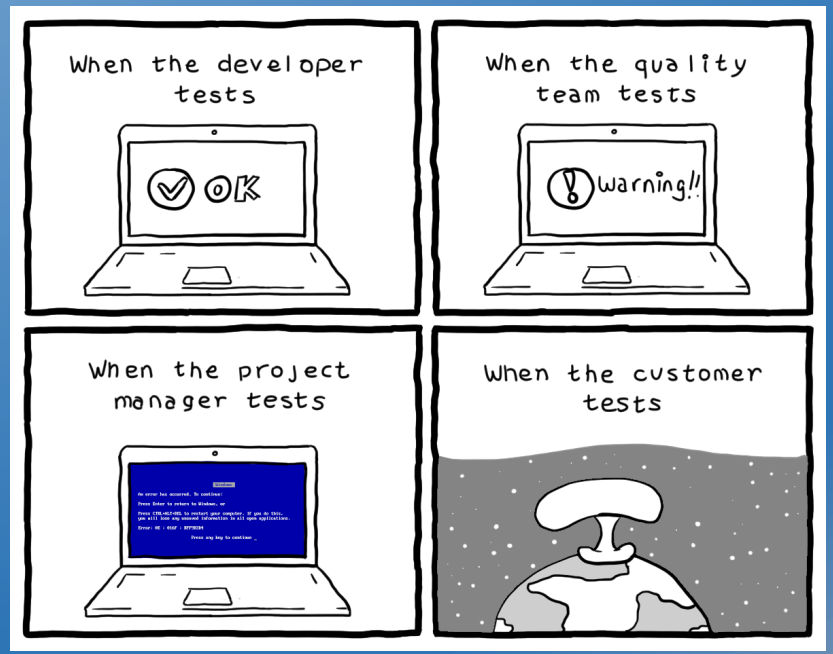




Testing I

CSCI2340: Software Engineering of Large Systems

Steven P. Reiss

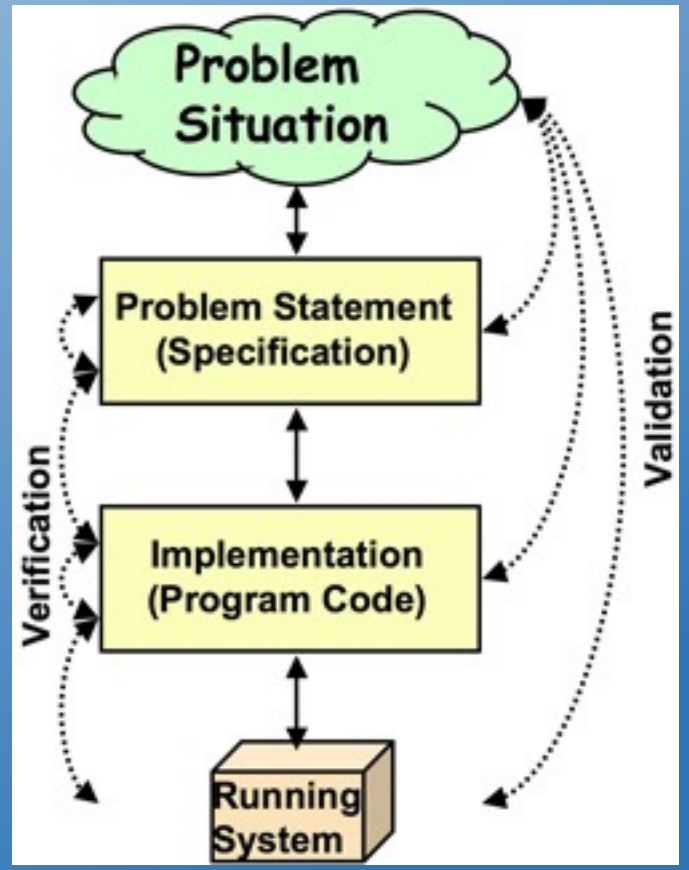


Grading

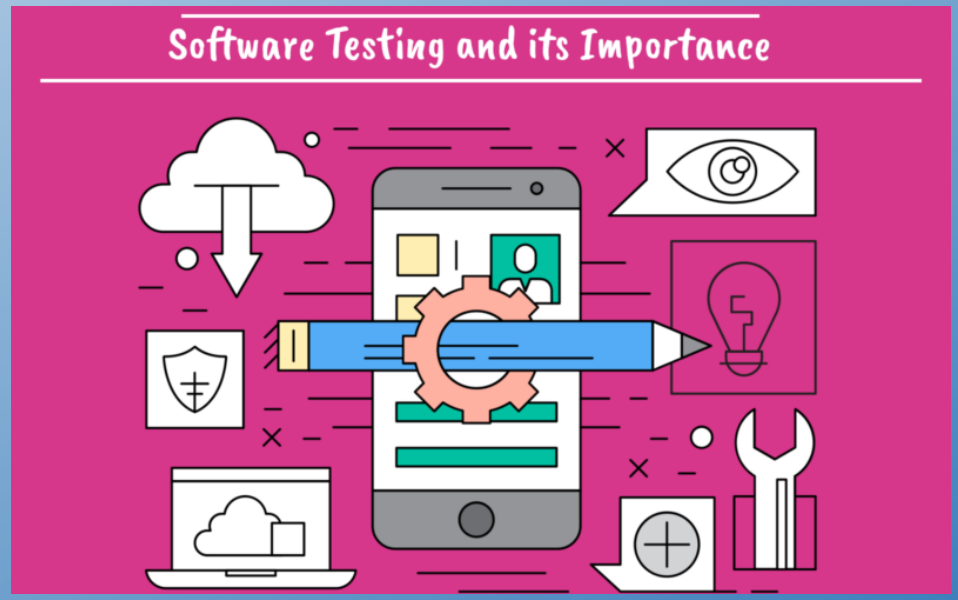
- Project Grading
  - Based on the various presentations
  - Based on reading code in the repositories
    - Code should meet the criteria specified in the course
    - Code should be designed to be usable many years in the future
    - Code should be designed for maintenance and evolution
  - Based on other items that should be in the repo
- Program Grading
  - Based on handing in all portions in in a timely fashion
  - Possibly on reading code as well

Verification versus Validation

- Formal verification
  - Proving that the program is correct
    - Proof checkers are getting much better
  - Requires a formal definition of correctness
    - This is often harder to write than the program itself
- Proving correctness of the code
  - Requires modeling the program (finite state)
    - Model might be in error or too abstract
  - Can be very difficult
    - Much can be automated, but not all
- One practical solution is testing
  - Another is partial correctness (covered after testing)



Importance of Testing



- Software Testing could be a course by itself
  - Terminology, tools, techniques
- Who should find the bugs
  - Programmer
  - Company (QA team)
  - End User
- Testing can help tell if your program works
  - What does “work” mean
    - Does the program do what it should?
    - Can the tester “break” the program?
    - Will users “break” the program?
  - Not whether it is correct
- Being your own tester
  - You should test all code before committing it
  - Force yourself (and your team) to use the system (a lot)
    - Dogfooding if appropriate

Limitations of Testing

Program testing can be used to show the presence of bugs, but never to show their absence

– E. Dijkstra in Structured Programming

- Testing does not define the program behavior
  - Doesn't provide a semantics for the program
  - Doesn't provide specifications (formal or otherwise) for the program
  - Doesn't consider everything involved in correctness
  - Doesn't consider all possible inputs
- Testing will tell you what is broken, not what works
  - Give you confidence in the program, not guarantees
  - Dijkstra: Testing can only show the presence of bugs, not their absence
- A successful test case is one that fails
  - Good testing tries to break the program, not validate it
  - Good testers are devious and good at breaking things
  - Assumption is that if it can't be broken, it mainly works

EXERCISE

Consider the following program. It takes three input data values representing the three lengths of the sides of a triangle as parameters. The method returns a string indicating whether the triangle is scalene (i.e., no two sides are equal), isosceles (two sides equal) or equilateral (all sides equal).

- Create a test set for this method
- Write down your tests

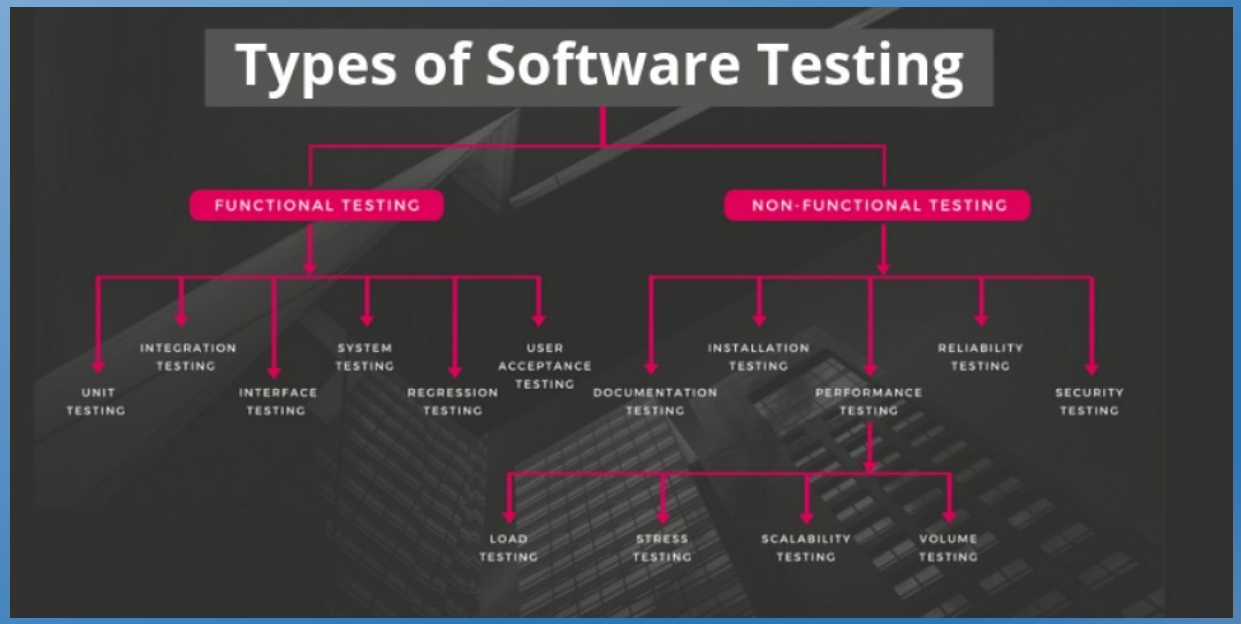
What Test Cases Did You Develop?

Discussion

- Do your tests include both the inputs and the expected output
- How many tests did you create?
- How many tests are needed to validate this program
  - 4 … 165 depending on who you ask
  - Depends on what you want to check, how much confidence you need
- Did you test for …
  - Valid cases (all three outputs)
  - All combinations of 2 sides being equal
  - Non-triangles (1,2,5)
  - Invalid inputs (0 or negative values, non-numbers (NaN), large numbers, …)
  - Approximate results (5.00000000000000001 vs. 5)
- And this is a trivial program …

Different Types of Testing

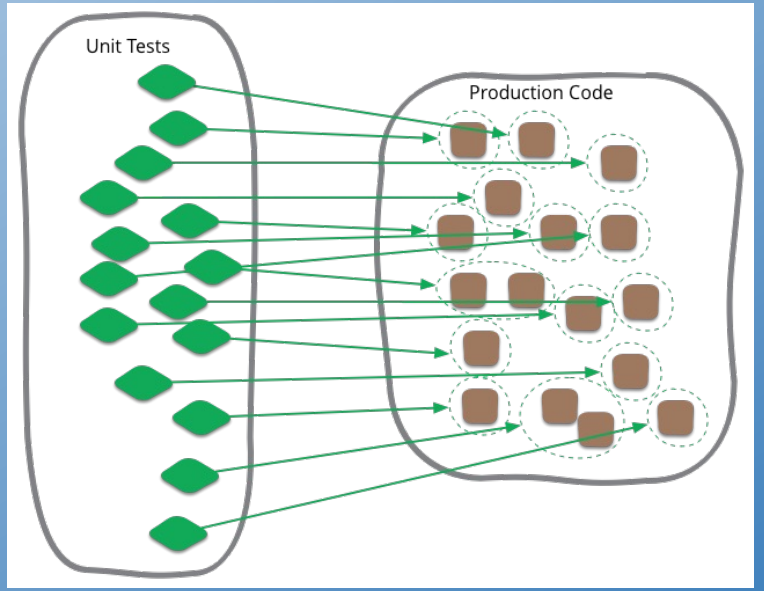
- Testing can be done at various levels
  - From individual function or method to system
- Testing can be done for different software aspects
  - Functionality
  - Security
  - Performance
  - Useability
  - Installation
  - Compatibility



Unit Testing

- Unit testing: methods or functions
  - Checking individual methods or functions
  - Junit for Java, similar systems for other languages
  - Sometimes worth doing for complex standalone methods
  - Worth doing for libraries and common code
  - Worth doing to prevent regressions (error history)
  - Difficult to do for much of an integrated system
- Unit testing: classes
  - Methods generally don't function outside of their class
    - Functions often don't function outside their file
    - Need to set up the class to test a method
  - Probably want to test multiple methods or functions at once
    - Add and remove from a structure…
  - Still can be difficult in an integrated system (classes rarely operate in a vacuum)

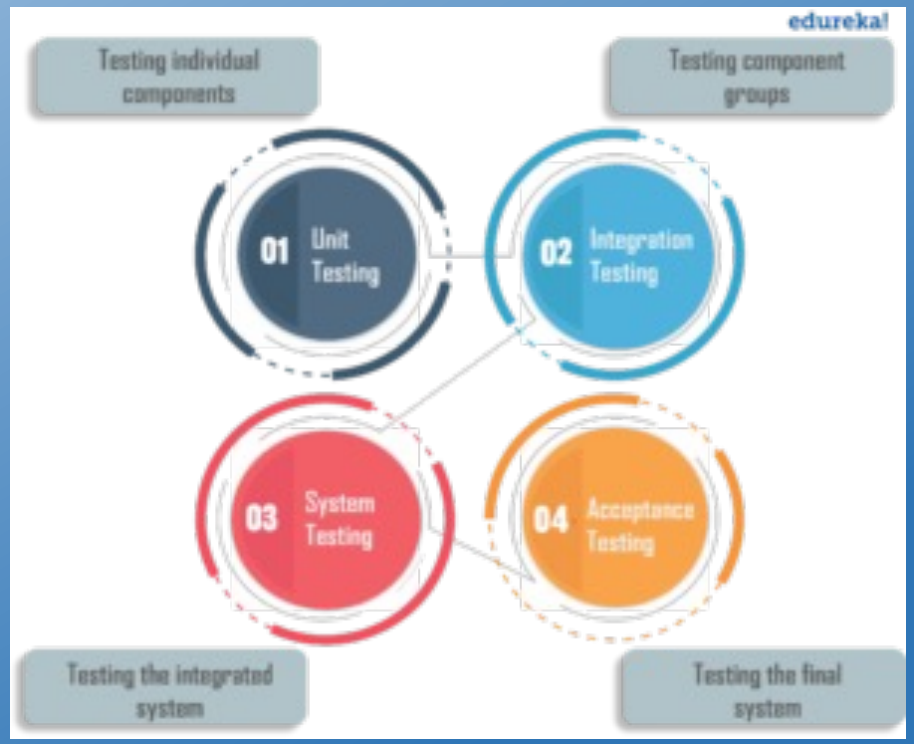
Unit Testing: Packages



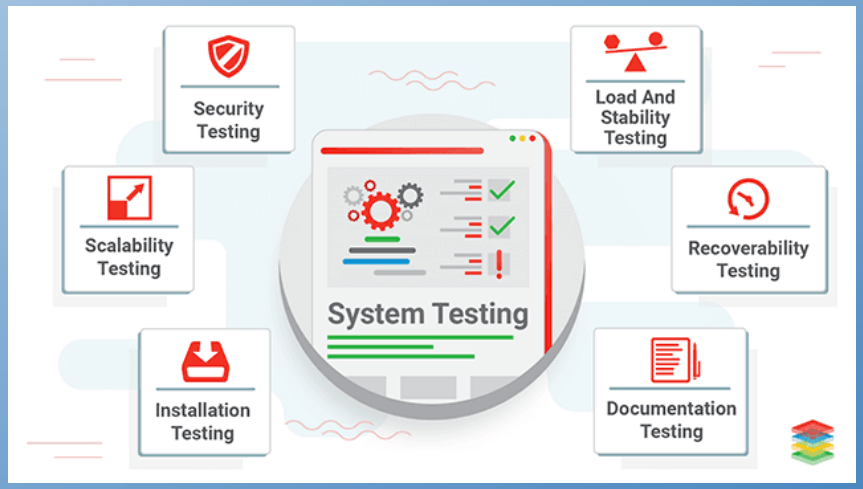
- Classes generally don't operate in isolation
  - They require other classes in the package and system
- Need to set up a whole environment to test them
  - Multiple objects of multiple types
  - Then you can test individual classes and methods
- Junit provides some hooks for this
  - @Before, @BeforeClass
  - @After, @AfterClass
- Provide a set of tests for a particular package or directory
  - Including package, class, and method tests
  - Early on you need to test functionality without other packages
    - Mocking, stub implementations, tracing interactions
    - This is where an interface-based design is helpful
  - Should be done within the package (so methods don't have to be public)
    - Test class or set of test classes either in same directory or in parallel directory (maven)
- Useful for testing your individual components

Integration Testing

- Multiple people and packages need to work together
  - Lots of potential errors
    - Wrong interface assumptions
    - Different interface assumptions
  - Always takes longer than anticipated
- Testing the interactions between packages
- More focused on functionality than problems
  - Can be higher-level package tests with actual packages
  - But should include error conditions as well
    - Note that these should be handled by defensive coding
- Different from interface testing
  - Testing all the calls in an interface
  - Especially with RESTful interfaces
  - This can focus on finding problems
  - Generally included in package testing



System Testing



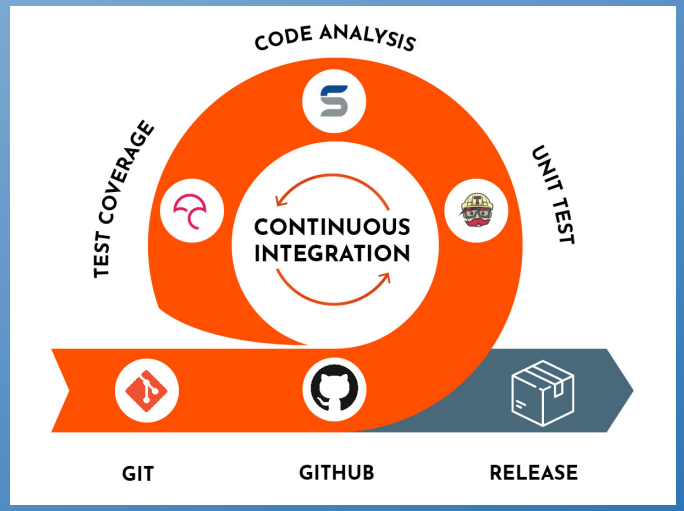
- Testing the whole system in operation
- Involves running through your scenarios
  - This is the minimum you need to do
  - Should include all scenarios, with their variations
  - Include error-handling with the scenarios
  - Automated if possible
- Additional tests based on
  - User reported problems
    - Often these can be view as unit tests (with some work)
  - Problems found during debugging
    - Again, these are often viewed as unit tests
  - Additional scenarios
    - Based on user experiences

Dogfooding

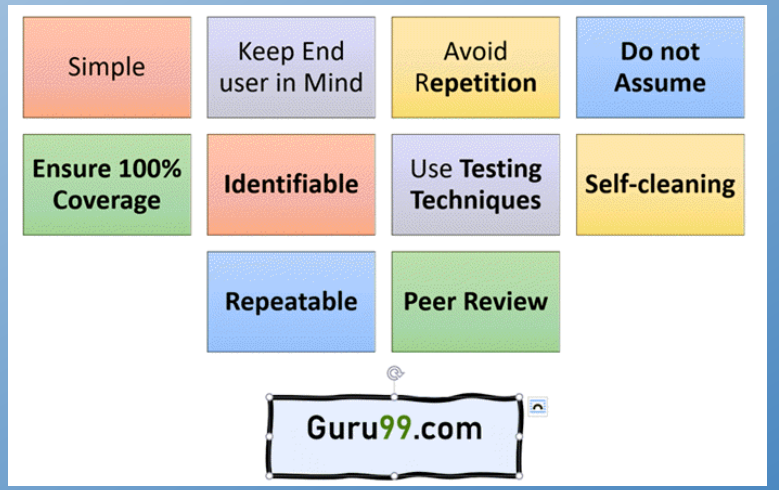
- Using what you are writing is a good habit to get into
  - Especially if you use it to assist in its development
  - Force yourself to use it even when it is a bit buggy
  - Provides testing without formal test cases
  - Provides experience
    - User experience (UX), performance, bugs, missing features, …
  - Provides tests of latest version of the system (prerelease)
  - Run your program from a debugger to catch bugs as they occur
  - Your projects can do this
    - Speech, IoT, Accessibility, LLAMA
    - UI Gen, DJ, Agentic
- Note that you won't be as critical as real users
  - You are too tolerant of your own mistakes
  - You know how difficult it might be to fix something
  - But you can use it when others cannot
  - Note potential problems even if you don’t fix them
    - Write them down so you remember them

Continuous Integration and Testing

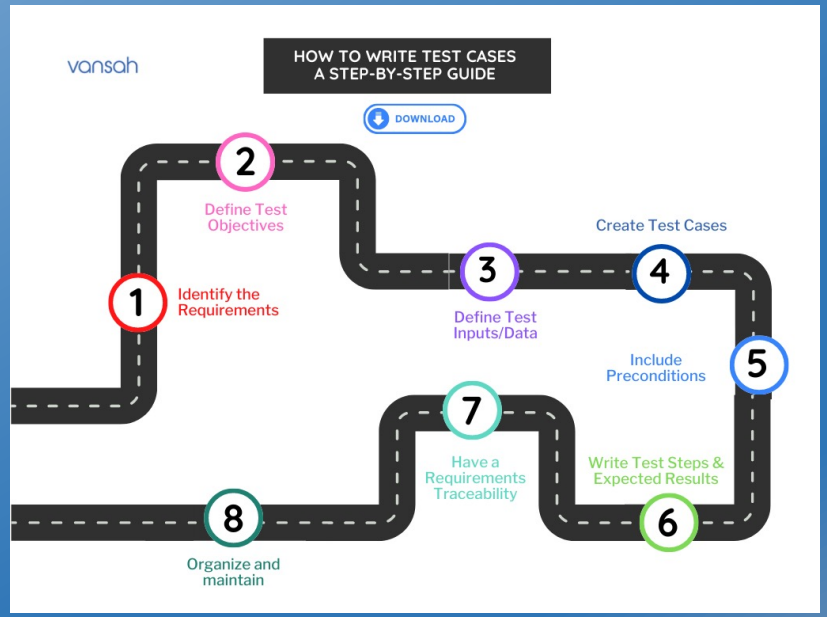
- We've talked about this before
  - Using a single branch
  - Merging on a regular (daily) basis
  - Running the experimental version of the system
- It also involves testing
  - Run all the test cases with each merge
  - Running tests is part of the merge process
  - Having an adequate set of tests is part of the process
- We'll look at this in more detail next time



Creating Test Cases



- Creating good tests is difficult
  - Can be as much work as writing the original code
  - Difficult to think of all possible tests
  - Difficult to ensure that the tests cover all possible cases
  - Difficult to check the results accurately
- Good test cases are designed to break the program
  - This is puzzle solving again
  - How can I break this program
  - What can I do that is unexpected
  - What are the cases that will cause problems
- Black box versus white box testing
- Tools are being developed to automate this
  - And best practices
  - Covered next time
- Problem: What is a GOOD set of tests?



Coverage

- How to measure the effectiveness of a test suite
  - Has it found all the bugs
  - Has it missed any obvious bugs
  - How confident in the program are you if it passes the tests
- A concrete measure of this is coverage
  - What part of the program is covered by the test set
    - Covered means executed by some test in the set
    - Any code not covered hasn't been tested
  - But what does coverage mean
- Other measures are also used
  - Coverage is only as helpful with quality tests
  - Mutation testing (next time)

Types of Coverage

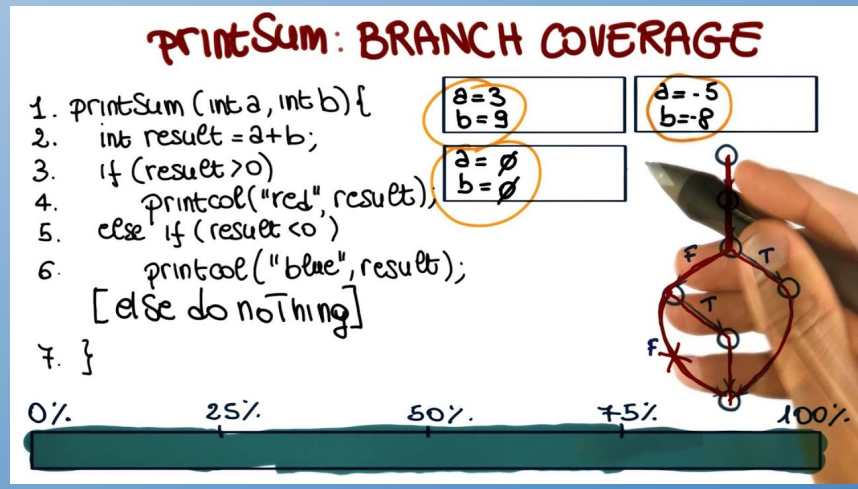
- Method coverage
  - Every method is executed by some test
  - Is this sufficient?
- Call coverage
  - Every method call is executed by some test
- Line coverage
  - Every line of code is executed by some test
  - Better than method coverage
  - Not always achievable (defensive coding)
  - Typically, this is what is thought of as coverage
    - What the coverage user interfaces show

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	221	84%	2970/3513	81%	859/1060	1.727
junit.extensions	6	82%	52/63	87%	7/8	1.25
junit.framework	17	76%	399/525	90%	139/154	1.605
junit.runner	3	49%	77/155	41%	23/56	2.225
junit.textui	2	76%	99/130	76%	23/30	1.686
org.junit	14	85%	196/230	75%	68/90	1.655
org.junit.experimental	2	91%	21/23	83%	5/6	1.5
org.junit.experimental.categories	5	100%	67/67	100%	44/44	3.357
org.junit.experimental.max	8	85%	92/108	86%	26/30	1.969
org.junit.experimental.results	6	92%	37/40	87%	7/8	1.222
org.junit.experimental.runners	1	100%	2/2	N/A	N/A	1
org.junit.experimental.theories	14	96%	119/123	88%	37/42	1.674
org.junit.experimental.theories.internal	5	88%	98/111	92%	39/42	2.29
org.junit.experimental.theories.suppliers	2	100%	7/7	100%	2/2	2
org.junit.internal	11	94%	149/157	94%	53/56	1.947
org.junit.internal.builders	8	98%	57/58	92%	13/14	2
org.junit.internal.matchers	4	75%	40/53	0%	0/18	1.391
org.junit.internal.requests	3	96%	27/28	100%	2/2	1.429
org.junit.internal.runners	18	73%	306/415	63%	82/130	2.155
org.junit.internal.runners.model	3	100%	26/26	100%	4/4	1.5
org.junit.internal.runners.rules	1	100%	35/35	100%	20/20	2.111
org.junit.internal.runners.statements	7	97%	92/94	100%	14/14	2
org.junit.matchers	1	9%	1/11	N/A	N/A	1
org.junit.rules	20	89%	203/226	96%	31/32	1.444
org.junit.runner	12	93%	150/161	88%	30/34	1.378
org.junit.runner.manipulation	9	85%	36/42	77%	14/18	1.632
org.junit.runner.notification	12	100%	98/98	100%	8/8	1.162
org.junit.runners	16	98%	321/327	96%	95/98	1.737
org.junit.runners.model	11	82%	163/198	73%	73/100	1.918

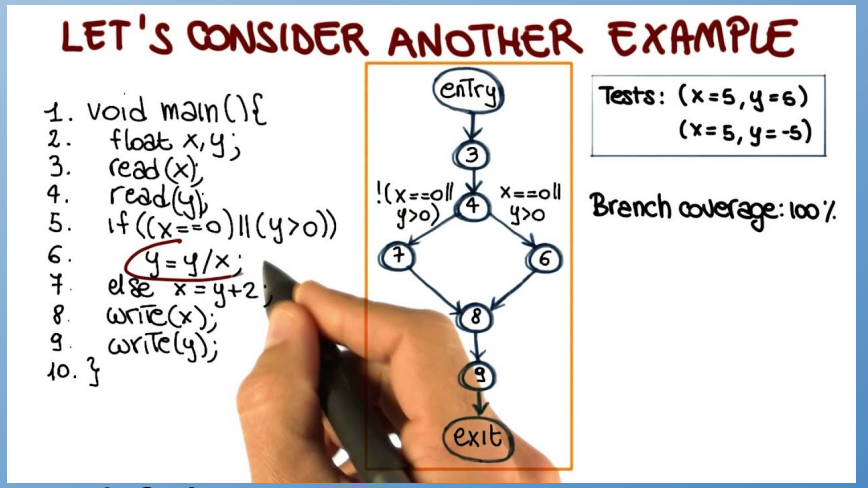
Report generated by Cobertura 1.9.4.1 on 12/22/12 2:25 PM.

Types of Coverage



- Statement coverage
  - Every statement in the code is executed by some test
  - Handles cases of multiple statements on a line
    - if (<condition>) <statement>
      - condition false implies line coverage, not statement coverage
- Branch coverage
  - Every alternative in each branch is executed by some test
    - if (<condition>) <statement>
      - condition true implies statement coverage
      - Branch coverage implies a test with true and a test with false

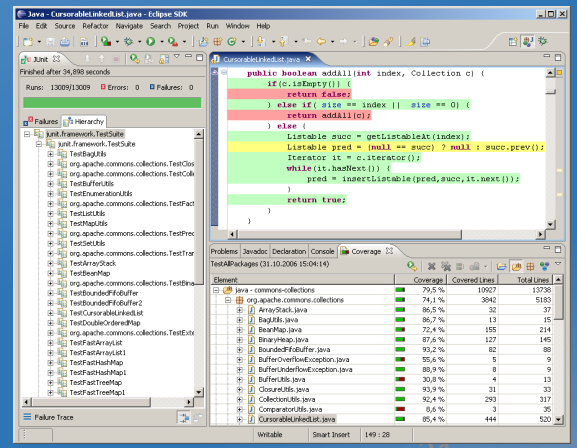
Types of Coverage



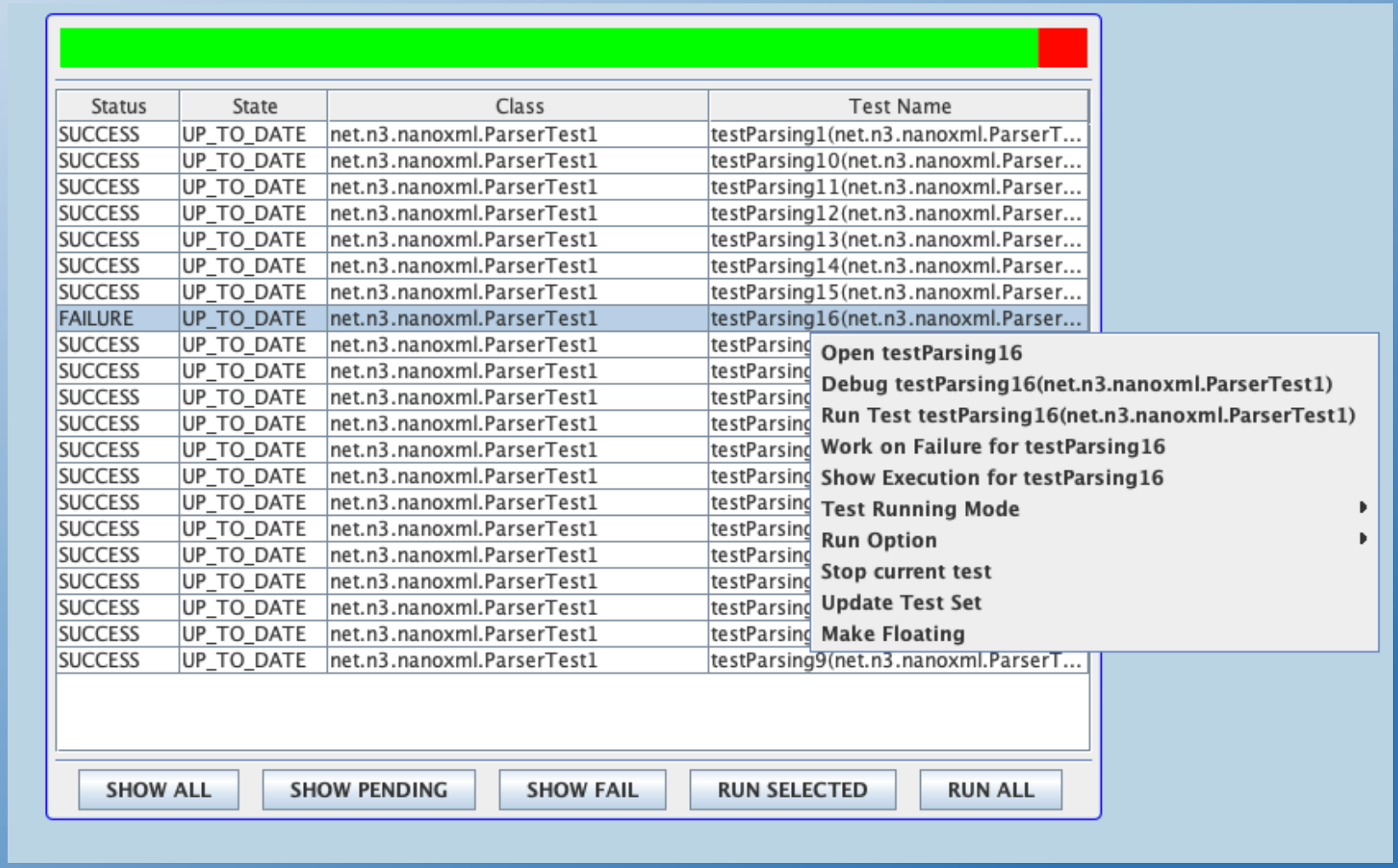
- Condition coverage
  - Every condition in a branch is covered with true and false
  - Handling && and || conditions
    - if (x && y && z)
      - Four cases: x false; x true, y false; x,y true, z false; x,y,z true
  - Statement coverage at the assembly level
- Path coverage
  - Every path through the program covered by some test
  - Sequences of conditions
    - Ten sequential independent if statements might yield 1024 paths
  - Talked about, but rarely used

Getting Coverage Information

- Most IDEs can provide coverage information while testing
  - Eclipse, IntelliJ collect coverage data (line, branch/condition)
    - You must run for profiling, not for debugging
    - VS Code has plugins that can do this
  - Provide a user interface to show covered lines
  - prof and gprof for C/C++ programs (outside of IDE)
- Code Bubbles automatically provides coverage for tests
  - Computes line, branch, call coverage
  - Uses line coverage for fault localization
  - Uses coverage to note when tests should be rerun
  - No current user interface otherwise
- Code Bubbles Test management



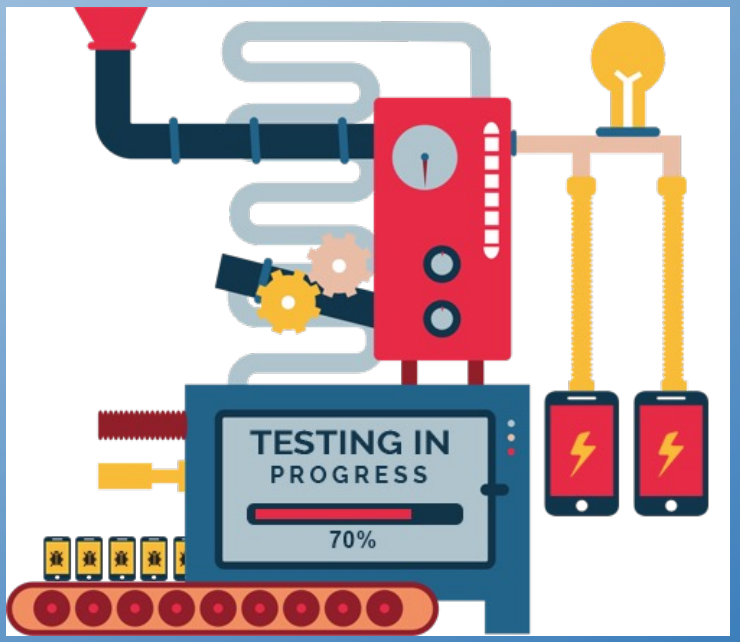
Code Bubbles Test Management



User Interface Testing

- Testing the user interface
  - Testing if the functionality works
  - Testing the appearance of the interface
  - Testing usability of the interface
- Testing with actual users
  - Dogfooding
  - Alpha and beta testing
  - A-B testing
- Testing functionality with simulated interaction
  - Test case runs an Input script
  - Tool support: Selenium and similar packages
- Testing different platforms (browsers, mobile platforms, window sizes, …)
- Testing accessibility and internationalization
- Testing installation and platform compatibility

Classic User Interface Testing



- Lab-based testing
  - Recruit (and pay) potential users
  - Train them on the system
  - Have them use it in a lab setting
- Record what they do
  - Have them talk through what they are doing
  - Save video and transcript
  - Analyze behavior (errors, confusion, # clicks, timings, …)
- Survey the users after
  - To understand what they did and why
  - To get other's opinions on the user interface

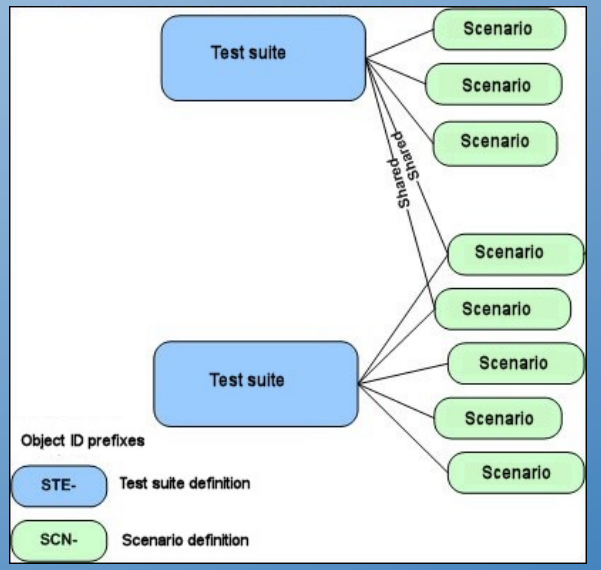
Continuous User Testing

- Gather information from a running system
  - Example: command sequences, # errors, # undos, …
  - Example: faults that are hidden from the user
  - Example: timings and performance
- Can be augmented with questionnaires
  - Simple: do you like/dislike the system
  - More advanced: full questionnaire with feedback
- Dogfooding as a form of continuous testing
  - But you need to note and report errors
- A-B testing
  - Some users run version A, some run version B (chosen at random)
  - Get feedback, monitor errors, timings, etc.
  - Need a significant user base

Test Maintenance

- Tests need to be kept up-to-date
  - Need to evolve as the code evolves
  - Need to be augmented as new problems are found
- If you don't run tests all the time
  - The tests eventually become useless
  - Or require more work than they are worth
- Maintaining tests can be a lot of work
  - If you have lots of tests
  - If the tests weren't written well and documented
  - If you haven't checked the tests for a long time
- Evolve tests as you evolve code
  - This is implied by continuous integration

Test Suites

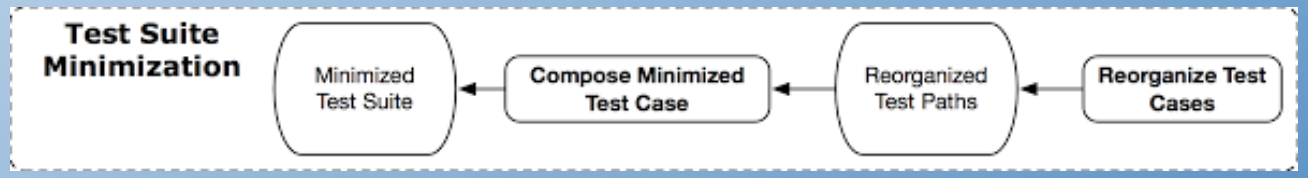


- A test suite is a set of tests
  - All the tests for all components of the system
  - Junit : run all tests, all tests in a class, a single test
- Problems with test suites
  - Serious testing can yield large test suites: 1000's of tests
  - Running all of these can take hours or days
  - Check the results of all the tests can be difficult

Test Prioritization

- Problem: Running a test suite can take hours or longer
  - Lots of tests, some tests can be time consuming
- One approach is to order or prioritize the tests
  - Test that are likely to fail are done first
  - Can be based on what code has changed
    - Tests that cover that code should be tried first
    - For some changes, this might be large
  - Can be based on other criteria
    - E.g., tests that failed recently should be tried first
- Only run tests affected by changes
- This has been an area of active research

Test Minimization



- Minimize the size of the test suite
  - Coverage provided by multiple tests can overlap
  - Doesn't mean the tests are identical
    - But often means the tests are redundant (not always)
- Find a minimal set of tests that achieve the same coverage
  - Same as the original set
  - Based on type of coverage one is trying to achieve
- This is NP-complete; but good approximations exist
- This has been an area of active research

Programming Assignment

- I want to do a code review next class
  - If you might like your program reviewed
    - Please send it to me (pointer to a repo is good enough)
  - I will select one submission
    - Make sure it is anonymous
    - And provide feedback on it
  - Do this today at some point (or before the weekend)
- If I don’t get anything suitable, I will pick one at random

PROJECT

- Should have individual components working next week
  - Want to have a minimal system running fairly soon
  - Status reports were encouraging in this line
- Make sure you have the framework for testing
  - Include a script to run all the tests
    - (ant or maven or gradle); inside or outside IDE
- Add information on how to compile and test
  - Should be noted in README files in your repository