Lecture Notes
November 28, 2017

JP Morgan coaches will be in class on Thursday

Second Iteration Development and Code Inspection due this Thursday November 30 (includes preliminary demo for second iteration)

Next week: review on Tuesday, second exam on Thursday

NO EXAM DURING FINALS WEEK

Remaining assignments:

Continuous Integration and Coverage due Friday December 8

JP Morgan Showcase Demos due Tuesday December 12

Setup time 2:30-3, science fair style demos 3-5, awards 5-5:30, refreshments 5-6:30

Final Delivery due Friday December 15

Final 360-degree Feedback due Saturday December 16 (do not submit until *after* everything else has been submitted!)

Penetration Testing

Typical security goals:

- Confidentiality: data not leaked
- Integrity: data not modified
- Availability: data is accessible when needed
- Authenticity: data origin cannot be spoofed

"White hat" 3rd party typically hired to perform pen testing - adversarial thinking

Threat model - who is the adversary and what actions can they perform?

- Black box to simulate external hacker
- Grey box to simulate malicious insider (or outside attacker who acquires credentials), increases the "surface area" of the test

Simulated security attacks, partially automated but usually manually intensive

- Reconnaissance - gather information about the target before the test
- Identify possible entry points
- Attempt to break in and access sensitive data
- Report findings

Besides determining security weaknesses, pen testing can also be used to test an organization's security policy compliance, its employees' security awareness and the organization's ability to identify and respond to security incidents

Targeted testing is performed by the organization's IT team and the penetration testing team working together, called "lights-turned-on" approach because everyone can see the tests being carried out

External testing targets a company's externally visible servers or devices including domain name servers (DNS), email servers, Web servers or firewalls - objective is to find out if an outside attacker can get in and how far they can get in once they've gained access

Internal testing mimics an inside attack behind the firewall by an authorized user with standard access privileges, useful for estimating how much damage a disgruntled employee could cause

Blind testing strategy simulates the actions and procedures of a real attacker by severely limiting the information given to the person or team that's performing the test beforehand - might only be given the name of the company (expensive!)

Double blind testing carries blind test a step further, only one or two people within the organization might be aware a test is being conducted - useful for testing an organization's security monitoring and incident identification as well as its response procedures

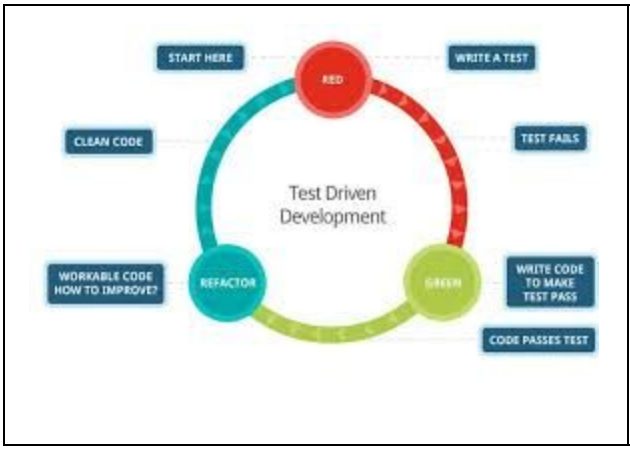
See OWASP Web Application Security Testing Cheat Sheet (OWASP = Open Web Application Security Project)

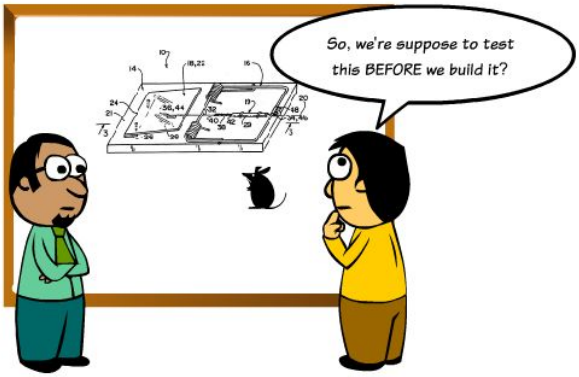
OWASP Top 10

OWASP Mobile Top 10

Test-driven development (TDD)

Red -> Green -> Refactor





Basic idea is to write each test case *before* writing the code to implement

Then the test cases for the new feature should initially *fail*… if a new test case already passes, something is wrong with the test case since the new feature doesn't exist yet!

In many cases the new test case will not even build, since it calling a method or using a class that doesn't exist yet - so first step after failing build is to create stubs for all dependencies - but the stubs shouldn't "do" anything yet

After the new test builds and runs, but fails, then write just enough code, and no more, to make the test pass

Then re-run the entire test suite, since your new code may have broken previous tests

If broken, fix - before continuing to additional new tests

Rinse repeat - there should be multiple test cases for the new feature, each very simple, using equivalence classes and boundary conditions as in conventional testing

Writing just enough code to make a test pass may result in hardcoded constants and other smelly code

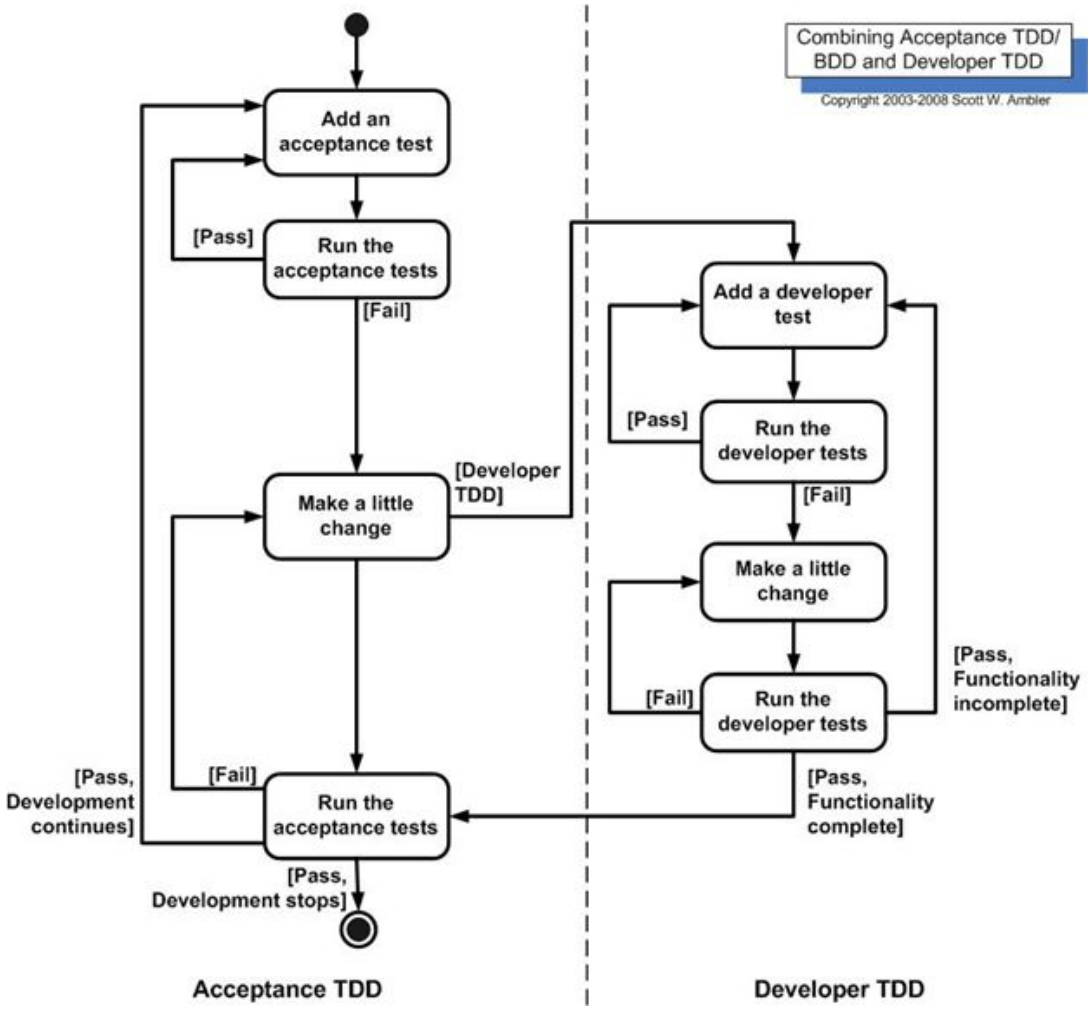
So refactor as soon as a test passes, *before* introducing the next test case

Initial test cases are system level, assuming the feature is visible at the system level, then unit level

> Generally, system level test cases cannot pass until some of the unit level tests pass, because there are no units (or changes to existing units) that implement the feature

System level TDD is sometimes called acceptance test driven-development (ATDD) or behavior-driven development (BDD), and unit level TDD may be called developer TDD

TDD should, in principle, result in 100% statement/branch coverage, since every statement and every branch condition is written to pass a specific test case that had previously failed



Goal of TDD is specification instead of validation

Thus TDD is a requirements technique – the set of test cases defines what the feature should do (conditions of satisfaction)

TDD is also a design technique – thinking about how to fulfill the system level test cases requires determining the data types and subroutines

Example: remember our requirements guessing game from September, where the customer wants us to develop a duck simulator?

Say we've developed the class diagrams shown in SimUDuck, but haven't started coding yet. What do we do next to perform test-driven development? What should our test cases be? How will they operate?

Duck sounds

Refactoring

The Head First book mentions “refactoring” many times but only presents one page in an appendix

Refactoring is a systematic approach to removing (non-comment) code smells

Modify code structure in very small steps that do not change any functionality (semantics-preserving)

Need to re-run affected test cases (or entire test suite) after every small change to verify that there were no functional changes

Sometimes the term “refactoring” is used to refer to *any* re-engineering of the code base even if behavior is not strictly preserved, e.g., replacing a data structure or algorithm, reorganizing a monolithic architecture into microservices

Basic idea is to clean up the code to make it easier to maintain (change)

Ideally, exactly the same test cases should apply and exactly the same test cases should pass

But if any of the changes cross unit boundaries, or introduce new units/remove old units, then we cannot expect the exact same unit tests to pass - may need new unit tests and even new system level tests (i.e., refactor the test suite)

Refactoring tools sometimes suggest code changes or even automate the code, often implemented as an IDE plugin (e.g., JetBrains’ PyCharm)

When to refactor: before fixing a bug, after fixing a bug, before adding a feature, after adding a feature – include refactoring time as part of bug or feature user story (this appears to add time but usually saves time later on)

Example refactoring patterns: see https://sourcemaking.com/refactoring