4156 November 29, 2016

Upcoming:

Thursday December 1st – Come to class prepared with questions for pre-exam review. Read "Head First" chapters 7-12 (and both appendices) and Ron Patton chapters 5-7 if you haven't already.

Thursday December 1st - Second Iteration Development and Code Inspection due, includes informal demo for team mentor (submit *after* demo)

Tuesday December 6th – Second exam

Thursday December 8th – Last day of class, discuss some advanced topics plus some in-class time for team meetings

Thursday December 8th – Continuous Integration and Coverage assignment due (submit in time for deadline, do **not** wait until showcase demo to submit)

Monday December 12th – short "showcase" demos, signup form coming soon

Friday December 16th – Final Delivery with final complete demo (submit *after* demo)

Tuesday December 20th – Third (final) 360-degree feedback due

NO EXAM DURING FINALS WEEK

Today's topics – refactoring, test-driven development

Refactoring:

Early in semester, we discussed "code smells"

Code smells might involve deviations from coding conventions, but that is not the main concern

Coding conventions can be checked very cheaply by static analysis tools known as "style checkers" or, much more expensively, by human code inspection – static analysis includes any processing of the code that does not involve running the code (any processing that involves running the code is called dynamic analysis)

Coding conventions are NOT typically checked or enforced by a compiler or interpreter, which only enforces language syntax – there can be multiple mutually incompatible coding conventions for the same programming language

Some IDEs and code editors *enforce* coding conventions, so it may not be possible to save a source file containing code that violates predefined or configured coding conventions

Example coding conventions: UpperCamelCase class names, lowerCamelCase method names, only use spaces for indentation never tabs

Code smells are generally other problems with the code, beyond coding conventions, and may include problems with the comments

Example code smells: Useless comments, Long methods, long parameter lists, duplicated code, multiple distinct code that does *almost* the same thing, large conditional logic blocks, large classes, etc. (see https://blog.codinghorror.com/code-smells/)

Code smells do not necessarily imply bugs, but they make it more likely that bugs will be introduced when changes are made

Code smells can be detected by static analysis or by human code inspection, looking for known "anti-patterns" (bad comments can usually only be detected by humans)

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

Modifies 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

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 automate the code changes and sometimes suggest code changes without automating, often implemented as an IDE plugin (e.g., JetBrains' IntelliJ IDEA, PyCharm, RubyMine, WebStorm)

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 refactorings: see https://refactoring.guru/catalog

Test-driven development (TDD):

Basic idea is to write each test case *before* writing the code to implement (technically, TDD = test-first development plus refactoring)

TDD is easier when adding a new feature to an existing system than to do from scratch, since the UI and infrastructure already exists

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 compile, so first step after failing compilation is to create stubs to enable compilation - but the stubs shouldn't "do" anything yet

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 some previous test

If broken, fix the new/old code (or modify the old test) before continuing to additional new tests - this is impractical for very large regression test suites, which may take hours/days to run, although there are test suite reduction tools that can help determine which code in existing tests are "affected" by the new/modified code

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

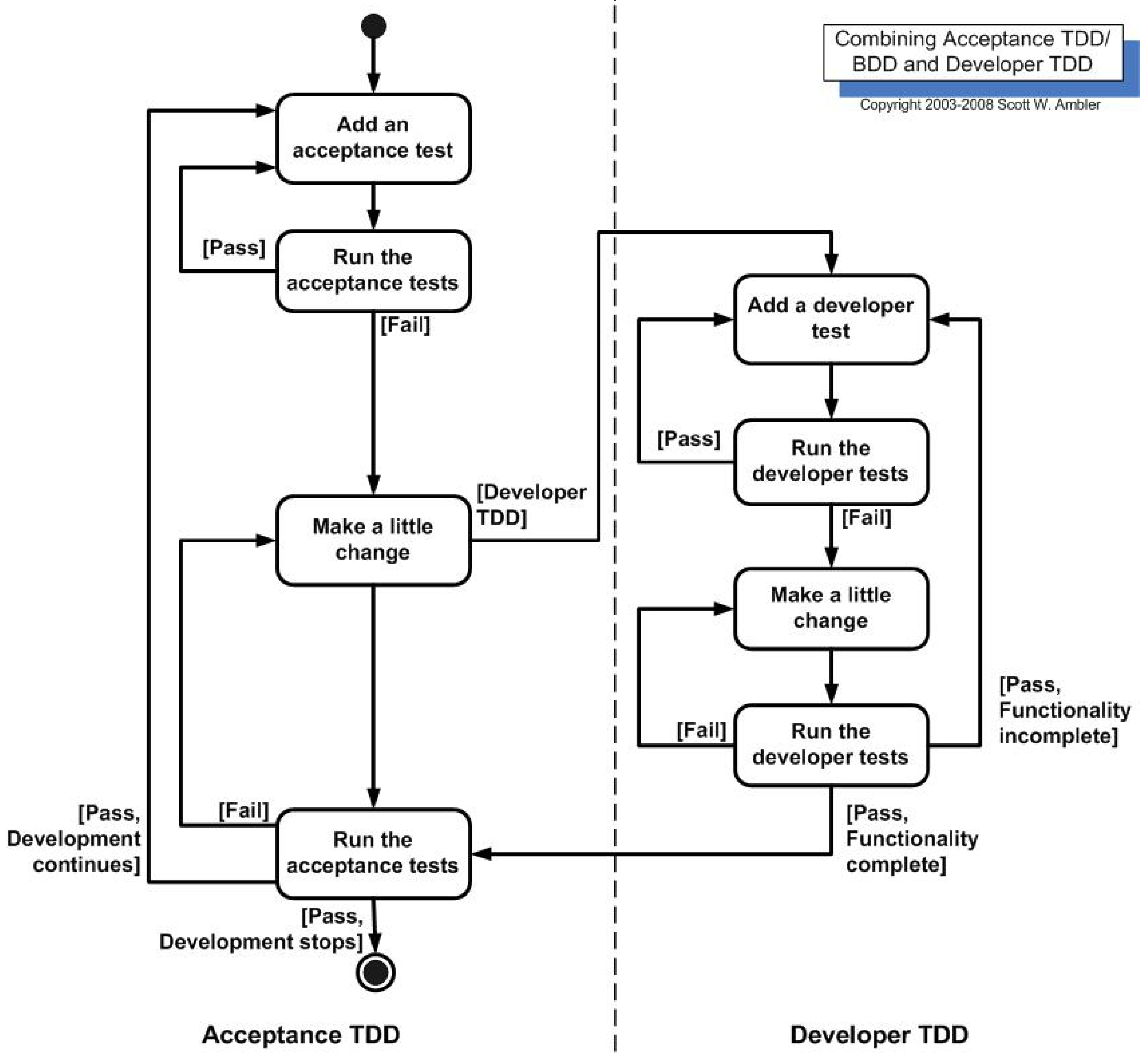
So refactor as soon as a test passes, before introducing the next test case

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) to 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 - recall “unit” means method, function, subroutine, procedure, etc.

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

In practice, may not be possible to test all UI, database, legacy, framework, third-party code



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