4156 December 1, 2016

Upcoming:

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 posted

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

Ron Patton ch. 5 "Testing the Software with Blinders On"

Dynamic blackbox testing at the system level – use the software as a customer would: enter inputs, receive outputs, check the results

Test case = specific inputs and procedures to follow (often manual)

Selecting tests cases is main concern – don't test too much, too little, or the wrong things.

Sometimes there isn't any spec – what should you do?

"Exploratory testing" treats the software itself as the specification. Methodically explore from the UI to determine the features.

Test-to-pass – First assure that software minimally works, don't push capabilities, don't try to break it. Only then test-to-fail (error-forcing is what most testing is for), strategically probing for weaknesses.

Equivalence partition (or equivalence class) – A set of prospective test cases that is likely to reveal the same bug, group similar inputs, similar outputs, similar operation of the software.

Reduce infinite possibilities to manageable effective set.

Example: For entering a filename, need valid characters, invalid characters, valid filename lengths, too short names, too long names.

Key concepts: boundary conditions, sub-boundary conditions, nulls, bad data.

Programming inherently susceptible to problems at edges. Typically programmer gets it right for vast majority of values in the middle but makes mistakes at the edges.

Look for types representing various flavors of numerics: position, quantity, speed, location, size, and similarly for characters.

Characteristics include first/last, start/finish, empty/full, slowest/fastest, largest/smallest, next-to/farthest-from, min/max, over/under, shortest/longest, soonest/latest, highest/lowest.

Test on both sides of a boundary, not just at the boundary: add one to maximum and subtract one from minimum.

Examples: Text entry field allows 1 to 255 characters. Try 1 character, 255 characters, both valid, then also 254 valid, 0 and 256 invalid.

If program reads/writes media, try saving minimal size file, maximal file, empty file, too big file.

Sub-boundary conditions (internal boundaries): Memory management sizes (powers of 2), ascii/unicode table.

Byte, word, double used internally by software although may not be explicitly visible to end-user. If word is N bits, try to overflow into a value that requires N+1 bits.

Ascii A-Z is 65-90 and a-z is 97 to 122. Consider characters mapped to just below, just above, and in between the alphabetic characters.

Although most software now uses Unicode, a given application may use an older version of Unicode with some unassigned values. Latest is from June 2016 http://www.unicode.org/versions/Unicode9.0.0/. Unicode now covers numerous language scripts (http://www.unicode.org/charts/) and emoji (http://www.unicode.org/emoji/charts/emoji-versions.html).

Default, empty, blank, null, zero, none: Consider case where user deletes default value automatically included in UI. Software may follow different values for 0 (or other default) vs. for blank.

Garbage data: Letters in a numeric field, non-printing characters in a text field, dates in the far past or far future, etc.

State testing, etc.: The rest of this chapter will **not** be covered by the exam. (While state machines are a good way to specify many web and mobile applications, with each state corresponding to a "page" or "screen", they do not work very well for increasingly common single-page-applications. Material about load and stress testing was thrown in with state testing, even though its orthogonal.)

Ron Patton ch. 6. "Examining the Code"

Static whitebox testing: examine and review code without running it.

"Formal review" = prepare, follow rules, identify problems, write report. Improves communications, quality, team camaraderie, solutions. Make sure to look for omissions as well as problems in the available code.

Increasing levels of formality:

"Peer reviews" = grab a buddy or two to look over your code. Be careful this is a real review, not a coffee break.

"Walkthrough" = author presents code during meeting with handful of participants, line by line or function by function, explaining what code does and why. Reviewers receive code in advance to prepare (examine, write comments and questions).

"Inspection" = Highly structured. Reader is not the author. Other participants, called inspectors, assigned to review from different perspectives – user, tester, customer support. One person might review backwards, from end to beginning, to make sure everything covered. Recorder takes notes. Moderator make sure rules are followed, verifies changes afterwards (if severe another inspection is required), and writes report.

Coding Standards and Guidelines: standards *must* be followed, guidelines are suggested best practices (although some companies do not distinguish, they are all mandatory).

Improve reliability, readability/maintainability, portability.

The chapter says "style" is individual idiosyncrasy and should not be checked, but many companies have mandated styles as well.

Generic Code Review Checklist and the remainder of the chapter will **not** be covered by the exam. (Much of this material is oriented to C/C++ and much of it doesn't make sense for modern languages.)

Ron Patton ch. 7 "Testing the Software with X-Ray Glasses"

Dynamic whitebox testing aka "structural testing", because you use the underlying structure of the code to design and run tests.

Although presented as part of dynamic whitebox testing, *all* dynamic testing that finds a bug should try to narrow down to the simplest test case that demonstrates the bug.

Unit testing vs. integration testing vs. system testing – Bugs found at the unit testing level must be in that unit. Bugs found during integration testing must involve mismatches between units and across unit interfaces. i.e., how those units interact, assuming all the units have already been testing.

Integration testing proceeds incrementally testing pairs of units together and/or adding one unit at a time, making it easier to isolate bugs.

Bottom-up integration testing: Write test drivers to exercise the collections of units. Works for pretty much any integration.

Top-down integration testing: Write stubs to act like the rest of the code being invoked by the collection of units being exercised. Obviously only works when the integration invokes other code (in addition to itself needing to be invoked).

Blackbox testing is usually done before whitebox testing, but whitebox analysis of the code can sometimes be used to reduce the number of Blackbox tests - because sometimes test cases expected to be treated differently (different equivalence partitions) is actually treated the same by the code.

Dataflow testing tracks a piece of data completely through the software under test (SUT). At the unit level this is just the module or function, but can be done with a set of integrated units or for the entire application. Could use a debugger and watch variables to view this data as the program runs, to check intermediate values during execution, but then do need to know which variables this data passes through (via static or dynamic analysis).

Can also use debugger to set values of variables to test cases that cannot be executed directly. Useful for enforcing all error messages and predefined exceptions to occur. But make sure the test case is valid, e.g., don't set value to N after the code explicitly checked that it was not N on all possible paths to this point.

Whitebox enables determining other sub-boundaries beyond powers-of-two and ascii/Unicode. For example, numerical analysis computation might switch between formulas at some value for better numerical precision. More generally, look for equations/formulas in code and try to force error conditions, e.g., divide by zero.

The main use of whitebox testing that we've discussed in past is code coverage. Need access to code to check that all code has been exercised. But also need a tool that will record as each is executed, so you can check what hasn't been executed. Then need to write specific test cases to force execution of the unexercised code.

Can also determine that some test cases are redundant, because they don't exercise any additional code.

The main metrics for coverage are statement (or line) and path. Branch, the simplest form of path coverage, is better than statement because it considers how the execution got to each statement. Condition is better than branch because it considers both true and false for every decision contributing to a branch.

100% coverage does not mean there are no bugs, because still have to consider different data values along the paths covered, but its better than <100% coverage. Condition > Branch > Statement.

Head First ch. 7 "testing and continuous integration"

Build (compilation) after every code change isn't good enough, also need to make sure the test suite still passes. "Continuous integration" runs all the test cases whenever the code is checked into the version repository (or on a fixed schedule such as nightly).

If your software doesn't work, it won't get used. Need to test from every perspective:

Users see the system from the outside, closed blackbox, only concerned with functionality.

Testers peek under the covers a little, checking data in the database, network connections are closed, memory usage is steady. Called "greybox testing".

Developers are responsible for everything, open whitebox.

Blackbox functionality testing is the most important testing. User input validation is also very important, with bad data rejected in a way that a typical end user can understand. Error messages are typically the last thing developers think about but the first thing users notice. Boundary cases and other off-by-one errors are the most likely to occur.

Greybox testing is 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. The sharing might be between applications within a mobile device or between users in the backend.

Verify auditing and logging (and that where it goes is secured), data destined for other systems, system-added information like checksums, hashes, timestamps, make sure there are no scraps left lying around (security risk as well as resource leak).

Make sure to place any resources (e.g., data sets) used in testing under version control.

Similar whitebox material to Patton, but also mentions thread-safe and security roles. And when to stop trying to increase coverage (by, say, 5% at a time): when you stop finding new bugs. (Or perhaps very rarely find new bugs, diminishing returns.) Track number of bugs found over time.

Frameworks provide a “test runner” to run your test cases, consistently, but you still have to define/write the test cases. (There are some automatic test generation tools, primarily concerned with security or performance issues, or increasing coverage, but they don’t and can’t know anything about the functionality that needs to be tested.)

Regression testing comes for free if you have a one-command testing framework that runs all your tests, because then you can run those tests after every change to detect “regressions” (which is when new code breaks old test cases).

However, 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.

One very important concern not mentioned by Patton is associating setup/teardown code with every test case (or with every interrelated set of test cases = test class). The setup code should assume a "clean" application state, and then put the system into the state necessary to conduct the test(s), and the teardown code should always restore memory, file system, network, database, etc. to "clean", with no traces of the test case that just completed. This may require using mocks rather than real resources, which are expensive to run and expensive to cleanup (and not always possible to cleanup because of external side-effects and visibility).

The absolutely most important reading to re-read (for the purposes of the second exam, and also for your future lives as professional software engineers) is chapter 7 of "Head First"! Which, btw, mentions the 3-to-1 testing to production code ratio I'd previously discussed.

"Head First" ch. 8 "test-driven development"

This is what I covered on Tuesday: TDD and refactoring, e.g., the tests should always fail before you write the code that implements the new feature, and then you write the simplest code to get the test to pass.

Start with getting the new test case to compile. Resist the urge to add anything you might need in the future (YAGNI). Avoid dependencies. "Evolutionary design" from refactoring after each time you add that simplest code.

Red, green, refactor.

Tests drive the implementation, each test should verify only one thing, avoid duplicate test code, keep your tests and supporting code in a mirror directory of your source code (under version control), use mock objects.

The chapter discusses the "strategy pattern", but this and other design patterns will **not** be covered by the exam. "Mock object frameworks" also will **not** be covered by the exam. (These are both very important, but there wasn't time to cover in detail, so they will not be on the second exam.)

"Head First" ch. 9 "ending an iteration"

Assumes that the iteration did NOT include many activities that, in the real world, it would include – most notably system testing before delivery and iteration review.

Parallel iterations for developers and testers, fitting bug-fixing into iterations.

Bug tracker tool. Metrics and zero-bug-bounce.

Revisits standup meetings, burndown charts, project velocity.

"Head First" ch. 10 "the next iteration"

Planning the next iteration based on previous iteration's project velocity.

Revise priorities (customer) and time estimates (developers), new task board.

Using third-party code, taking over legacy code.

"Head First" ch. 11 "bugs"

More about using other people's code, pulling it into your process and repository.

Spike to estimate testing and debugging time.

"Head First" ch. 12 "the real world"

Reviews process concepts, task board, user stories, version control, CI, TDD, coverage.