Michael Bolton
DevelopSense

http://www.developsense.com

Why Test?

- When we test, we
  - try the product
  - to learn
  - sufficiently
  - everything that matters
  - about how the product *can* work
  - and how it *might* fail.

Test coverage is...

the extent to which we have traveled over some map of the system

Test Coverage Isn't *Code* Coverage

- *Code* coverage is only one way of modeling test coverage
  - and a fairly weak way, at that
- It might even be necessary, but it's definitely not sufficient
  - 100% code coverage might still miss all kinds of performance, reliability, or usability problems
  - code coverage doesn't cover missing features or functions
- Code coverage tools usually only cover *our* code
  - they don't cover calls to operating system and third-party code
- We don't, and often *can't*, know how variations (like varying the data or the platform) might cause us to take new branches in *other people's* code

Modeling Test Coverage

- To obtain better test coverage, we might
  - consider risks, and test for them
    - a risk is "a problem that might affect some person, caused by a vulnerability in the program, that is triggered by some threat"
  - test based on more detailed or varied structural models
    - what are the pieces of the product, and how do they interact with each other?
    - how does the product interact with other systems?
  - test more of the functions and capabilities of the system
    - what does the product do?
  - vary the data that we use or produce in our tests
    - what do the functions process?

Modeling Test Coverage

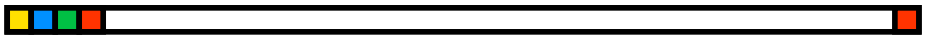
- To obtain better test coverage, we might
  - vary the platforms that we use in our tests
    - a platform is "anything upon which our product depends which is outside the scope of our current development project"
  - vary the operational modes of the system
    - operational modes are "ways in which the system might be used"
  - vary the time dimension
    - the ways in which our product works over time
    - the ways in which time affects our product or the other models
    - spend more time testing

Test Session Effectiveness

- A "perfectly effective" testing session is one entirely dedicated to test design, test execution, and learning
  - a "perfect" session is the exception, not the rule
- Test design and execution tend to contribute to test coverage
  - varied tests tend to provide more coverage than repeated tests
- Setup, bug investigation, and reporting take time away from test design and execution
- Suppose that testing a feature takes two minutes
  - this is a highly arbitrary and artificial assumption—that is, it's *wrong*, but we use it to model an issue and make a point
- Suppose also that it takes ten minutes to investigate and report a bug
  - another stupid, sweeping generalization in service of the point
- In a 90-minute session, we can run 45 feature tests—*as long as we don't find any bugs*

How Do We Spend Time?

(assuming all tests below are *good* tests)



Module	Bug reporting/investigation (time spent on tests that find bugs)	Test design and execution (time spent on tests that find no bugs)	Number of tests
A (good)	0 minutes (no bugs found)	90 minutes (45 tests)	45
B (okay)	10 minutes (1 bug, 1 test)	80 minutes (40 tests)	41
C (bad)	80 minutes (8 bugs, 8 tests)	10 minutes (5 tests)	13

Investigating and reporting bugs means….

SLOWER TESTING or…

REDUCED COVERAGE …or both.

• In the first instance, our *coverage* is great—but if we're being assessed on the number of bugs we're finding, we look bad.
• In the second instance, coverage looks good, and we found a bug, too.
• In the third instance, we look good because we're finding and reporting lots of *bugs*—but our *coverage* is suffering severely. A system that rewards us or increases confidence based on the number of bugs we find might mislead us into believing that our product is well tested.

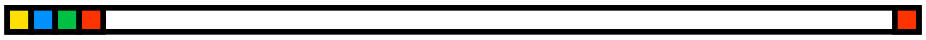
8

In the first instance, our coverage looks great—but if we're being assessed on the number of bugs we're finding, it looks bad. In fact, if we haven't found any bugs, maybe it *is* bad. The numbers on their own don't tell that story. In the third instance, we look good because we're finding and reporting lots of bugs—but our coverage is suffering severely. A system that rewards us or increases confidence based on the number of bugs we find might mislead us into believing that our product is well tested. In the second instance, coverage *looks* good, and we found a bug, too. But maybe our coverage isn't so good; maybe we've exercised a lot of very similar test ideas.

This is a powerful argument for testability. Testability includes: ♦ scriptable interfaces to the product, so that we can drive it more easily with automation; ♦ logging of activities within the program; ♦ real-time monitoring of the internals of the application via another window, a debug port, or output over the network; ♦ simpler setup of the application; ♦ the ability to change settings or configuration of the application on the fly; ♦ clearer error/exception messages, including unique identifiers for specific points in the code, or WHICH file was not found, thank you; ♦ availability of modules separately for earlier integration-level testing; ♦ information about how the system is intended to work (ideally in the form of conversation or "live oracles" when that's the most efficient mode of knowledge transfer); ♦ information about what has already been tested (so we don't repeat some else's efforts); ♦ access to source code for those of us who can read and interpret it; ♦ improved readability of the code (thanks to pairing and refactoring); ♦ overall simplicity and modularity of the application; ♦ access to existing ad hoc (in the sense of "purpose-built") test tools, and help in creating them where needed; ♦ proximity of testers to developers and other members of the project community; ♦ and finally, an application that's in good shape to start with, thanks to diligent testing by programmers, based on unit tests or (perhaps better yet) a test-first development approach such as test- or behaviour-driven development. You may have some of these things already; few projects implement all of them. Pick one that you're missing, and start there.

What Happens The Next Day?

(assume 6 minutes per bug fix verification)



Fix verifications	Bug reporting and investigation today	Test design and execution today	New tests today	Total over two days
0 min	0	45	45	90
6 min	10 min (1 new bug)	74 min (37 tests)	38	79
48 min	40 min (4 new bugs)	2 min (1 test)	5	18

Finding bugs today means….

VERIFYING FIXES LATER

…which means….

EVEN SLOWER TESTING or…

EVEN LESS COVERAGE …or both.

- …and note the optimistic assumption that all of our fixed verifications worked, and that we found no new bugs while running them. Has this ever happened for you?

9

With a more buggy product

- More time is spent on bug investigation and reporting
- More time is spent on fix verification
- Less time is available for coverage

With a *less* buggy product...

(that is, one that has had some level of testing already)

- We've got *some* bugs out of the way already
- *Those* bugs won't require investigation and reporting
- *Those* bugs won't block our ability to test more deeply

Test Early and Often!

- Recurrent themes in agile development (note the small A)
  - test-first development
  - automated unit tests
  - testability hooks in the code
  - automated builds and continuous integration
- The ideas are
  - to increase developers' confidence in and commitment to what they're providing ("at least it does *this*")
  - to allow rapid feedback when it *doesn't* do *this*
  - to permit robust refactoring
  - to increase test coverage and/or reduce testing time

Testing vs. Investigation

- Note that I just gave you a compelling-looking table, using simple measures, but notice that we still don't really know anything about…
  - the quality and relevance of the tests
  - the quality and relevance of the bug reports
  - the skill of the testers in finding and reporting bugs
  - the complexity of the respective modules
  - luck

…but if we *ask better questions*, instead of letting data make our decisions, we're more likely to *learn important things*.

We Testers Humbly Request...

- Provide testability
  - log files
  - scriptable interfaces
  - real-time monitoring capabilities
  - configurability
  - access to “live oracles” and other forms of information
    - to avoid wasting time investigating a “bug” that isn’t a problem
- Test at the unit level
  - use TDD, test-first, automated unit tests, reviews and inspections, step through code in the debugger—whatever increases your own confidence that the code does what you think it does