Is Unit Testing Worthwhile?

Nima Seyedtalebi

University of Kentucky

November 7, 2018

Background

- ➤ The IEEE Software Engineering Body of Knowledge (SWEBOK) provides a concise definition of software testing: "Software testing consists of the *dynamic* verification that a program provides *expected* behaviors on a *finite* set of test cases, suitably *selected* from the usually infinite execution" [11]
- Key points:
 - Dynamic: Input and source code are not always enough to determine behavior
 - Expected: We must be able to define expected behavior to test for it
 - Finite: The set of possible test cases is practically infinite, so we must choose a finite subset
 - Selected: Test cases can vary in usefulness considerably, so the choice is important

Different Kinds of Testing

- ► Testing can be classified by target or objective
- Classifying by target gives three levels:
 - Unit Testing: Small pieces of software testable in isolation
 - ▶ Integration Testing: Interactions between software components
 - System Testing: An entire system
- Classifications by objective:
 - Regression testing
 - Acceptance testing
 - Security testing
 - Performance testing
 - Stress testing

What is Unit Testing?

- ► From the SWEBOK: "Unit testing verifies the functioning in isolation of software elements that are separately testable." [11]
 - ▶ What constitutes a unit? It depends on context
 - Developers may have differing ideas about what constitutes a unit
- Usually performed by the developer of the unit or someone with programming skills and access to the source code
- Surveys suggest unit testing is an important testing method that sees widespread use
- Unit testing is sometimes conflated with other kinds of testing
 - E.g. a "unit test" that relies on a database connection is not a unit test under the definition given

Challenges in Software Testing

- Tests that are written without referring to some external specification can only suggest that the code does what the developer intended
- Exhaustive testing is impractical at best and impossible at worst. Consider a program like "echo" in Unix that takes a Unicode string argument:
 - ▶ With Unicode 11, 137374ⁿ permutations of length *n* are possible[?]
- ► Some tests are more useful than others. How do we choose the best set of tests?
- How do we know if we have enough tests?
- How do we know if testing is effective?
- ► Testing always involves a trade-off. More tests may find more problems, but tests take time to write and maintain

Common Techniques for Choosing a Test Set

- ▶ Ad-hoc: Choose test inputs based on intuition and experience
- Boundary-value Analysis: Choose inputs close to boundaries in the input domain e.g. largest and smallest possible values for numerical datatypes
- Code-based analysis techniques:
 - Control Flow Analysis: Choose tests that follow a subset of the possible control flow paths through the code
 - ▶ Data Flow Analysis: Choose tests that follow a subset of the possible data flow paths through the code
 - Mutation Analysis: Choose tests that fail when the program under test is changed slightly
- The code-based techniques are often used to assess test sufficiency

Software Metrics

- Failure: An undesired behavior
- Fault: The cause of a failure
- ► Defect: A fault or failure
- General software measures:
 - Code size (lines of code)
 - Number of independent paths through code (cyclomatic complexity)
 - Degree of nesting
 - Average number of parameters
 - Coupling Between Objects (CBO)
 - ► Fan-out, or how many classes does this class use?
 - Fan-in, or how many classes use this class?
- Survey data are used to measure things that are difficult to measure objectively

Code Quality and Test Sufficiency

- Test quality, test sufficiency, and the quality of the software under test are distinct
- Measures of quality:
 - ► Fault classification, count, and density
- Measures of test sufficiency:
 - Coverage, often expressed as a percentage
 - Mutation score
- The number of defects detected by a test suite can serve as a measure of test quality or sufficiency
 - Quality: Test suite A is better than suite B because it found more bugs
 - Sufficiency: Test suite A is sufficient because it found at least n bugs

Control-Flow and Data-Flow Analysis

- Units contain assignment statements and conditional statements
- Units have well-defined entry and exit points
- A path is a sequence of instructions
- Conditional statements determine control flow
- Assignment statements determine data flow
- Control-flow and data-flow analysis both involve selecting tests so their execution follows different paths through the code
- They differ in perspective and how paths are selected:
 - Control flow analysis considers paths between the entry and exit points
 - Data flow analysis considers paths that start with an assignment statement and end with the last use of the variable

Coverage Metrics

- Coverage metrics assess how many execution paths are tested versus how many are possible
- Metrics are based on desired level of coverage
- More complete coverage means exploring a larger portion of the possible execution paths
- Path selection criteria for control-flow analysis:
 - Statement Coverage: All statements are executed at least once
 - ▶ Branch Coverage: Every branch is taken at least once
 - Predicate Coverage: Every combination of truth value for every conditional is tried at least once
 - ▶ All-Paths Coverage: Every execution path is tried at least once
- ► Path selection criteria for data-flow analysis are based on when variables are defined and used

Mutation Analysis

- ▶ Mutation score comes from mutation analysis, first proposed in a 1978 article "Hints on Test Data Selection" [4]
- Key insights:
 - Programmers usually write software that is "almost correct"
 - Finding simple errors uncovers complex errors
- Used to assess test data sufficiency
- Mutation analysis involves making small, syntactically-legal changes to the unit under test, producing mutants
 - If the mutant causes some test to fail, it is said to be dead
 - If the mutant does not cause any tests to fail, it is said to be alive, killable, or stubborn
 - Mutants are killed when the test set is sufficiently sensitive to detect the mutation
- Mutation score is the number of mutants killed divided by the total number of mutants



Arguments for Unit Testing

- Helps uncover defects early in the development process
- ► Allows developers to refactor with confidence because breaking changes will cause the tests to fail
- Can encourage good software design
 - Unit testing requires the unit under test (UUT) to be isolated
 - ► Tightly-coupled units require more effort to test
 - ► Tightly-coupled units are less robust
 - Difficulty or undue effort in testing indicates suggest code needs refactoring to reduce coupling
- ▶ Tests serve as a form of documentation

Arguments Against, or Unit Testing Considered Harmful

- Unit testing does not positively affect code quality in practice
 - Most tests only assess whether the code does what the developer intended
 - Developers write lower-quality code to meet coverage-based requirements
- Low-quality tests are worse than no tests at all since they must be maintained
- ▶ Unit tests provide a false sense of security
- Unit testing costs more time than it saves
- Integration and system testing are more effective at uncovering defects

What Does the Research Say?

- No correlation found yet between unit testing and code quality[6]
- No correlation found between coverage-based methods for determining test sufficiency quality and code quality[6]
- Developers need a better understanding of what makes a unit test good[3]
- ➤ Test-Driven Development (TDD), of which unit testing is an integral part, seems to measurably improve software quality in some cases[10],[7]
- ► Automated test generation is in use, but mostly used in cases where specifications are not required[3]

Limitations of Unit Testing

- Cannot detect faults in the interaction between units or between subsystems
- Add to the complexity of the software under development
 - ▶ Bad unit tests have costs beyond the time to write them
 - Good unit tests that are hard to maintain may be worse than no unit tests at all
- ► Any test can contain defects
- Cover only a tiny fraction of the state space

What About Test-Driven Development?

- ► Test-driven development (TDD) is a development style built around two rules:[1]
 - "Write new code only if you first have a failing automated test"
 - "Eliminate duplication"
- ► Important points:
 - ► Test: Unit tests are written before new code
 - Failing: The test must fail at first
 - Automated: Tools are used to run tests and collect results
- ▶ "The goal is clean code that works..." [1]
 - Clean code has the smallest possible number of dependencies
 - Empirical studies of TDD use different measures
 - The ambiguity was probably intentional¹

¹ Test-Driven Development By Example says," TDD is an awareness of the gap between decision and feedback during programming, and techniques to control that gap"

...and the Research?

- ► A 2008 article [7] found modest improvements in code size and complexity but not in coupling or cohesion
- ➤ A 2013 meta-analysis[10] of 27 empirical studies found that TDD "results in a small improvement in quality but results on productivity are inconclusive."
- ➤ A recent (July 2018) article[8] called "What Do We (Really) Know About Test-Driven Development?" offers the following:
 - Use of TDD is uncommon in practice
 - TDD does appear to improve some measures of quality in some cases
 - Evidence for the effects of TDD on productivity is inconclusive
 - ▶ The order of testing is not the important part of TDD
 - Using a short development cycle had much more impact on quality than the order of testing (test-first versus test-last)

Conclusions

- Unit testing can be worth the trouble, but it is not sufficient by itself to improve software quality
- An iterative development process with short, gradual steps seems to improve software quality
- Unit testing and TDD are tools. Like other tools, they work best in the hands of those that know how to use them
- ► Test quality is very important since we are limited to a very small subset of possibilities when testing
- Testing is a balancing act
 - ► Test selection is an important problem
 - ► Tests are not free to maintain, even if a machine writes them for you
- Automation and tool support is a poor substitute for thinking about design

So What Can Be Done?

- "A Survey On Unit Testing Practice and Problems" [3] suggests:
 - Developers need help identifying what to test and whether a given test is good or not
 - Automatic test generation helps with the "how" of testing but not the "what"
 - The question of "what" is shared across all types of testing
 - Tests should be realistic
- Furthermore:
 - Software design is a skill that must be learned and practiced
 - Though part of design, testing is a distinct skill that must be learned and practiced

References:



Test-Driven Development By Example.

Addison-Wesley Professional, 2003.

David Bowes, Tracy Hall, Jean Petrić, Thomas Shippey, and Burak Turhan.

How good are my tests?

In Proceedings of the 8th Workshop on Emerging Trends in Software Metrics, WETSoM '17, pages 9–14, Piscataway, NJ, USA, 2017. IEEE Press.

E. Daka and G. Fraser.

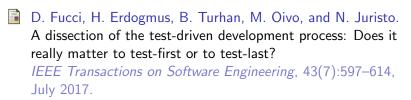
A survey on unit testing practices and problems.

In 2014 IEEE 25th International Symposium on Software Reliability Engineering, pages 201–211, Nov 2014.

R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer.

Computer, 11(4):34-41, April 1978.





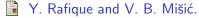
L. Gren and V. Antinyan.
On the relation between unit testing and code quality.
In 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pages 52–56, Aug 2017.

D. Janzen and H. Saiedian. Does test-driven development really improve software design quality? IEEE Software, 25(2):77–84, March 2008.

I. Karac and B. Turhan. What do we (really) know about test-driven development? *IEEE Software*, 35(4):81–85, July 2018.

Kshirasagar Naik and Priyadarshi Tripathy.

Software testing and quality assurance: theory and practice. John Wiley & Sons, 2011.



The effects of test-driven development on external quality and productivity: A meta-analysis.

IEEE Transactions on Software Engineering, 39(6):835–856, June 2013.

IEEE Computer Society, Pierre Bourque, and Richard E. Fairley.

Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0.

IEEE Computer Society Press, Los Alamitos, CA, USA, 3rd edition, 2014.