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" [4]
- 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." [4]
  - 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, 137374^n 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 Test Techniques

- 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
- Control Flow Analysis: Choose tests that follow a subset[1] of the possible paths through the code
  - Often defined in terms of coverage. A piece of code is "covered" if it executes at least once
  - Statement, branch, and decision/condition coverage are examples
  - Coverage is used as a measure of test sufficiency as well
- These techniques could also be considered different kinds of testing in some contexts

[1] not necessarily a proper subset

Software Metrics

- Failure: An undesired behavior
- Fault: The cause of a failure
- Defect: A fault or failure
- Measures of the program under test:
  - Fault classification, count, and density
- Measures of the test set:
  - Coverage, often expressed as a percentage
- General software measures:
  - Code size
  - Complexity
- Survey data are used to measure things that are difficult or impossible to measure objectively like perceived quality or ease of maintenance

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 Unit Testing

- 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[2]
- Coverage-based methods for determining test sufficiency are ineffective at improving software quality[2]
- Developers do not have a clear understanding of what makes a unit test good[1]
- Test-Driven Development (TDD), of which unit testing is an integral part, seems to measurably improve software quality[**?**],[3]

Limitations of Unit Testing

Unit tests:

- Cannot detect faults in the interaction between units or between subsystems
- Require some way of specifying the "correct" behavior of the unit
- Add to the complexity of the software under development
- Can contain defects

What About Test-Driven Development (TDD)?

- What is it?
- What are the benefits?
- How does TDD related to unit testing practice?
- How does TDD affect software quality?

Pitfalls to Avoid

-

There's Work to be Done!

-

Conclusions

-

References

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.

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.

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