“Quality is free, but only to those who are willing to pay heavily for it.”

— DeMarco & Lister
Your application is a special snowflake

Expert

Excuses for Not Writing Unit Tests

O RLY?

@ThePracticalDev
YOU ARE HERE

SPECIFY PRODUCT

SPECIFY SOFTWARE

CREATE SW ARCHITECTURE

DESIGN MODULES

IMPLEMENT

UNIT TEST

INTEGRATION TEST

SOFTWARE TEST

ACCEPTANCE TEST

PRODUCT

TRACEABILITY & VALIDATION

Test Plan & Test Results

Product Requirements

Software Requirements

High Level Design

Detailed Design

Source Code

Test Plan & Test Results

Test Plan & Test Results

Test Plan & Test Results

Test Plan & Test Results

Software Test Results

Integration Test Results

Test Results

Unit Test Results

Test Results

Test Results

Test Results
**Unit Testing**

- **Anti-Patterns:**
  - Only system testing
  - Testing only “happy paths”
  - Forgetting to test “missing” code

- **Unit testing**
  - Test a single subroutine/procedure/method
    - Use low level interface ("unit" = "code module")
  - Test both based on structure and on functionality
    - White box structural testing + Black box functional testing
  - This is the best way to catch corner-case bugs
    - If you can’t exercise them here, you won’t see them in system testing

---

**Test cases:**
\[
\begin{align*}
a &= 0; b &= 0; \\
a &= -1; b &= +1; \\
&\cdots \\
\end{align*}
\]

\[
\text{uint16_t proc(uint16_t a, uint16_t b)} \\
\{ \text{...} \\
\text{return(result);} \\
\}
\]

**Expected Test Results:**
\[
\begin{align*}
a &= 0; b &= 0; & \Rightarrow 0 \\
a &= -1; b &= +2; & \Rightarrow 1 \\
&\cdots \\
\end{align*}
\]
Black Box Testing

- Tests designed based on behavior
  - But without knowledge of implementation
  - “Functional” or behavioral testing

- Idea is to test what software does, but not how function is implemented
  - Example: cruise control black box test
    - Test operation at various speeds
    - Test operation at various underspeed/overspeed amounts
    - BUT, no knowledge of whether lookup table or control equation is used

  - Advantage: can be written only based on requirements
  - Disadvantage: difficult to exercise all code paths

https://goo.gl/wJeZ56
White Box Testing

- Tests designed with knowledge of software design
  - Often called “structural” testing
  - Sometimes: “glass box” or “clear box”

- Idea is to exercise software, knowing how it is designed
  - Example: cruise control white box test
    - Exercise every line of code
      - Tests that exercise both paths of every conditional branch statement
    - Test operation at every point in control loop lookup table

- Advantage: helps getting high structural code coverage
- Disadvantage: doesn’t prompt coverage of “missing” code
Boundary tests:
- At borders of behavioral changes
- At borders of min & max values, counter rollover
- Time crossings: hours, days, years, ...

Exceptional values:
- NULL, NaN, Inf, null string, ...
- Undefined inputs, invalid inputs
- Unusual events: leap year, DST change, ...

Justify your level of coverage
- Trace to unit design
- Get high code coverage
- Define strategy for boundary & exception coverage
**MCDC Coverage as White Box Example**

**Modified Condition/Decision Coverage (MC/DC)**
- Used by DO-178 for critical aviation software testing
- Exercise all ways to reach all the code
  - Each entry and exit point is invoked
  - Each decision tries every possible outcome
  - Each condition in a decision takes on every possible outcome
  - Each condition in a decision is shown to independently affect the outcome of the decision
- For example: “if (A == 3 || B == 4)” ➔ you need to test at least
  - A == 3 ; B != 4 (A causes branch, not masked by B)
  - A !=3 ; B == 4 (B causes branch, not masked by A)
  - A !=3 ;  B != 4 (Fall-through case AND verifies A==3 and B==4 are in fact responsible for taking the branch)

https://www.youtube.com/watch?v=DivaWCNohdw
**Lucy:** What’s modified condition / decision coverage testing?

**Philippa:** Here’s an analogy.

**Lucy:** Imagine a light controlled by three switches...

**Philippa:** In MC/DC testing, we need to show that each light switch can independently turn the light on or off...

**Philippa:** How does that apply to software?

**Philippa:** The light corresponds to the decision and the switches correspond to conditions.
Example MCDC

http://www.verifysoft.com/en_example_mcdc.html

- Each clause is tested with both true and false ("_" is don’t care)
- Each clause’s value (true/false), if inverted, would change value of output

by: \(( ((u == 0) || (x>5)) && ((y<6) || (z == 0))) \)

A full Test Coverage would consist into building the following truth table and testing each combination:

| Test case n° | A: (u == 0) | B: (x>5) | C: (y<6) | D: (z == 0) | \(( A || B) && (C || D) \) |
|--------------|-------------|----------|----------|-------------|---------------------------|
| 1            | F           | F        | F        | F           | F                         |
| 2            | F           | F        | F        | T           | F                         |
| 3            | F           | F        | T        | _           | F                         |
| 4            | F           | T        | F        | _           | F                         |
| 5            | F           | T        | F        | T           | T                         |
| 6            | F           | T        | T        | _           | T                         |
| 7            | T           | _        | F        | F           | F                         |
| 8            | T           | _        | F        | T           | T                         |
| 9            | T           | _        | T        | _           | T                         |

On the other hand, to ensure Modified condition/decision coverage, we should test (for instance) only the 5 combinations here-before underlined in yellow.

https://www.verifysoft.com/en_example_mcdc.html
Unit Testing Frameworks

- **Cunit as an example framework**
  - **Test Suite**: set of related test cases
  - **Test Case**: A procedure that runs one or more executions of a module for purpose of testing
  - **Assertion**: A statement that determines if a test has passed or failed

- **Test case example**: (http://cunit.sourceforge.net/doc/writing_tests.html#tests)

  ```c
  int maxi(int i1, int i2)
  { return (i1 > i2) ? i1 : i2; }
  ...
  void test_maxi(void)
  { CU_ASSERT(maxi(0,2) == 2); // this is both a test case + assertion
    CU_ASSERT(maxi(0, -2) == 0);
    CU_ASSERT(maxi(2,2) == 2); }
  ```

Best Practices For Unit Testing

- **Unit Test every module**
  - Use a unit testing framework
    - Don’t let test case complexity get too high
  - Use combination of white box & black box
    - Get good coverage, ideally 100% coverage
  - Get good coverage of data values
    - Especially, validate all lookup table entries

- **Unit Testing Pitfalls**
  - Creating test cases is a development effort
    - Code quality for test cases matters; test cases can have bugs!
  - Difficult to test code can lead to dysfunctional “unit test” strategies
    - Breakpoint debugging is not an effective unit test strategy
    - Using Cunit to accomplish subsystem testing is not really unit testing
  - Pure white box testing doesn’t test “missing” code

https://goo.gl/SjzaBm