Electrical & Computer ENGINEERING

18-642:

Unit Testing

9/18/2017

© 2017 Philip Koopman

Carnegie Mellon University

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:

```
a = 0; b = 0;
a = -1; b = +1;
...
```

```
uint16_t proc(uint16_t a, uint16_t b)
{ ....
  return(result);
}
```

Expected Test Results:

```
a = 0; b = 0;     ==> 0
a = -1; b = +2;   ==> 1
...
```

© 2016-2017 Philip Koopman

Black Box Testing

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

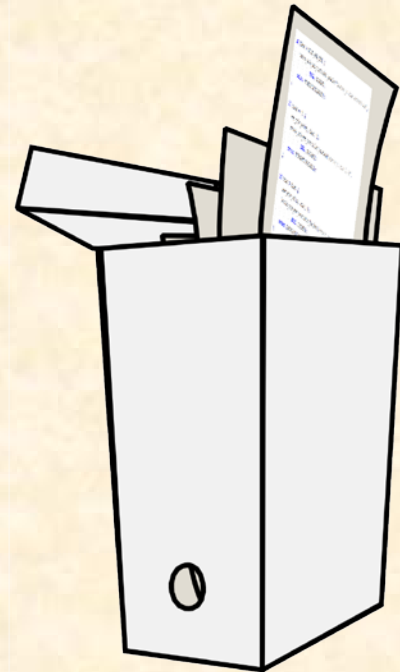
- **Idea is to test what software does, but not how function is implemented**
  - Example: cruise control black box test
    - Test operation at various speeds
    - Test operation at various underspeed/overspeed amounts
    - BUT, no knowledge of whether lookup table or control equation is used
  - Advantage: can be written only based on requirements
  - Disadvantage: difficult to exercise all code paths

© 2016-2017 Philip Koopman

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



© 2016-2017 Philip Koopman

Unit Testing Strategies

- **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

© 2016-2017 Philip Koopman

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)

MC/DC : EXAMPLE

a && b && c

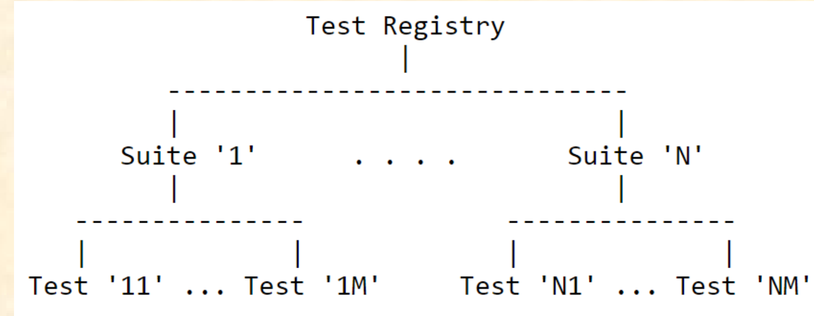
Test case	a	b		outcome
1	True	True	True	True
2	True	True	False	False
3	True	False	True	False
4	True	False	False	False
5	False	True	True	False
6	False	True	False	False
7	False	False	True	False
8	False	False	False	False
1	True	True	True	True
5	False	True	True	False

https://www.youtube.com/watch?v=DivaWCNohdw

© 2016-2017 Philip Koopman

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



http://cunit.sourceforge.net/doc/introduction.html

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

```
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); }
```

© 2016-2017 Philip Koopman

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

© 2016-2017 Philip Koopman