CS 4350: Fundamentals of Software Engineering
CS 5500: Foundations of Software Engineering

Lesson 5.3 Evaluating Tests

Jon Bell, John Boyland, Mitch Wand

Khoury College of Computer Sciences

Testing Evaluates Software Systems

- Validation: Are we building the right product?
- Verification: Are we building the product right?

How Do We Evaluate Tests?

- Purpose: Are tests checking the right things?
- Adequacy: Are they checking the things right?

Learning Objectives for this Lesson

- By the end of this lesson, you should be able to:
  - Describe measures of test suite adequacy, and to know their limitations;
  - Distinguish flaky and brittle tests;
  - Name several test smells, and give examples;
  - Explain some properties of good tests.

Review: Four Purposes of Tests

- Acceptance Test
  - Customer-level requirement testing
  - Validation: Are we building the right system ?
- Functional Test
  - "Black-Box" testing
  - Specification Testing
- Structural Test
  - "White-Box" testing
  - Exercising the code
- Regression Test
  - Prevent bugs from (re-)entering during maintenance.

These purposes are copied from Lesson 5.1

Adequacy of Acceptance Tests

- Crucial: meet with prospective customers.
- This is difficult, time-consuming and expensive.
- But building the wrong product is much worse!

Supplement to Acceptance Evaluation

- *Dogfooding* ("Eat your own dogfood")
- Be your own customer.
- Weaknesses:
  - Employees unrepresentative of customers
  - Whether someone can be compelled to use a product does not say whether they would purchase it.

Foreshadowing

- In Lesson 6.1, we cover "User-Centered Design"
- These techniques can help us generate and evaluate acceptance tests.

More later!

Functional Testing Adequacy

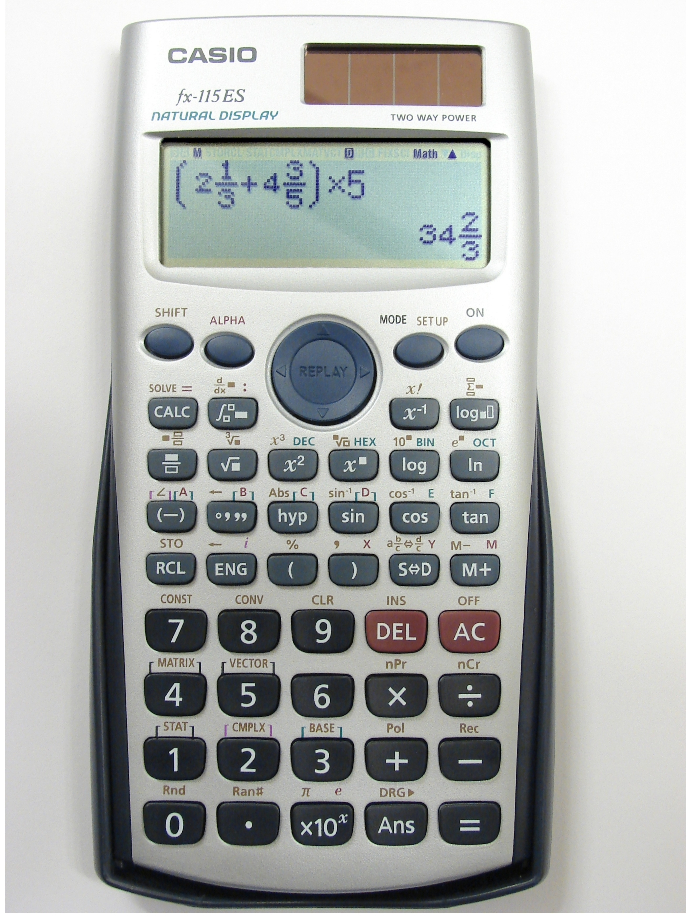
- Functional Tests also known as “Black-Box” testing.
- Testing without regard to the implementation.
- Functional tests are proxies for a *specification*:
  - A precise definition of all behavior of a SUT (outputs, state mutation, other effects) in all situations (state and inputs)
  - A specification may be formal (mathematical), informal (natural language) or implicit (“I know it when I see it”).
  - Adequacy of test suite is probability that an implementation passing all the tests actually fulfils the specification.

Not coverage of the SUT space!

E.g.: If a test contradicts the specification, the suite including it has zero adequacy!

Coverage of Abstraction of SUT (1)

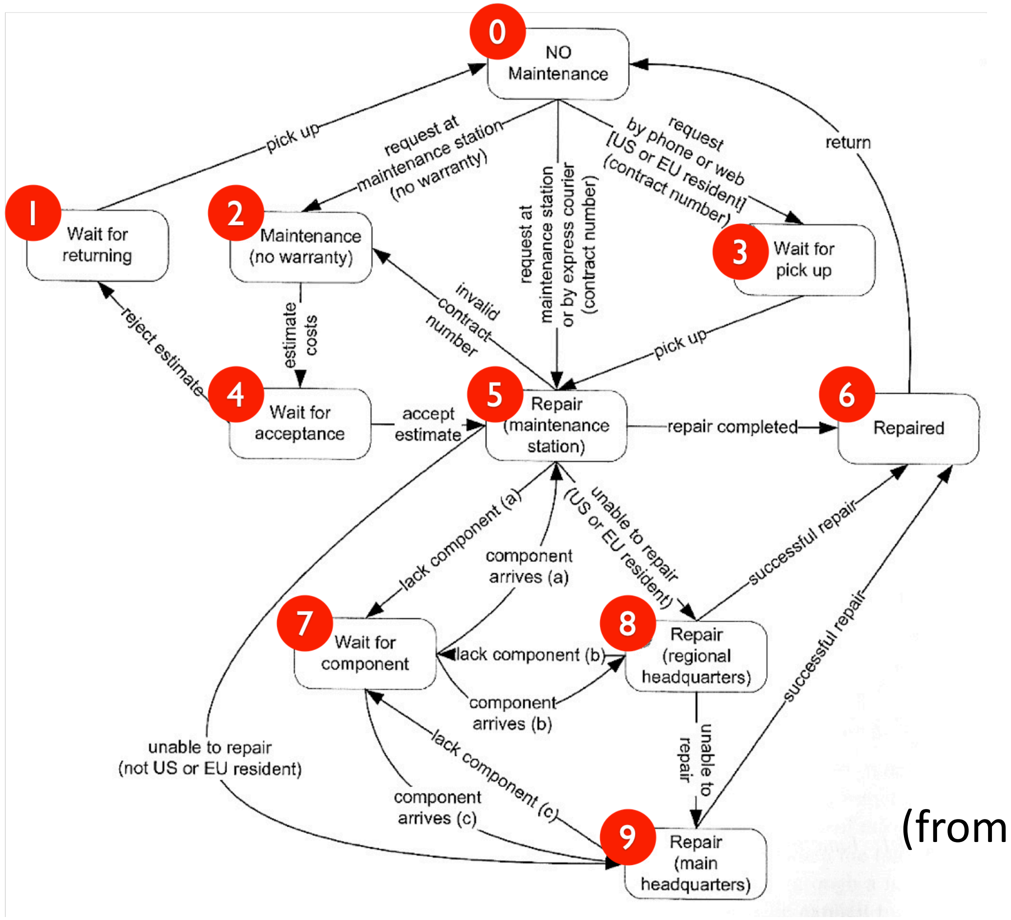
- Find independently testable features (ITFs)
  - Test these separately;
- Convert Cartesian product of possibilities to sum;
- Danger: missed interaction



Coverage of Abstraction of SUT (2)

- Select “special” values out of a range
  - Boundary values;
  - Barely legal, barely illegal inputs;
  - Ignore others;
- Integer overflow a serious problem: may be implicit
  - ComAir problem due to a list getting more than 32767 elems
- https://arstechnica.com/uncategorized/2004/12/4490-2/

Coverage of Abstraction of SUT (3)



- Abstract specification as a DFA
- Then use *Structural Testing* over the abstraction.
- Danger: system may be more complex than the model.

(from Pezze + Young, “Software Testing and Analysis”, Chapter 10)

Adequacy of Structural Testing

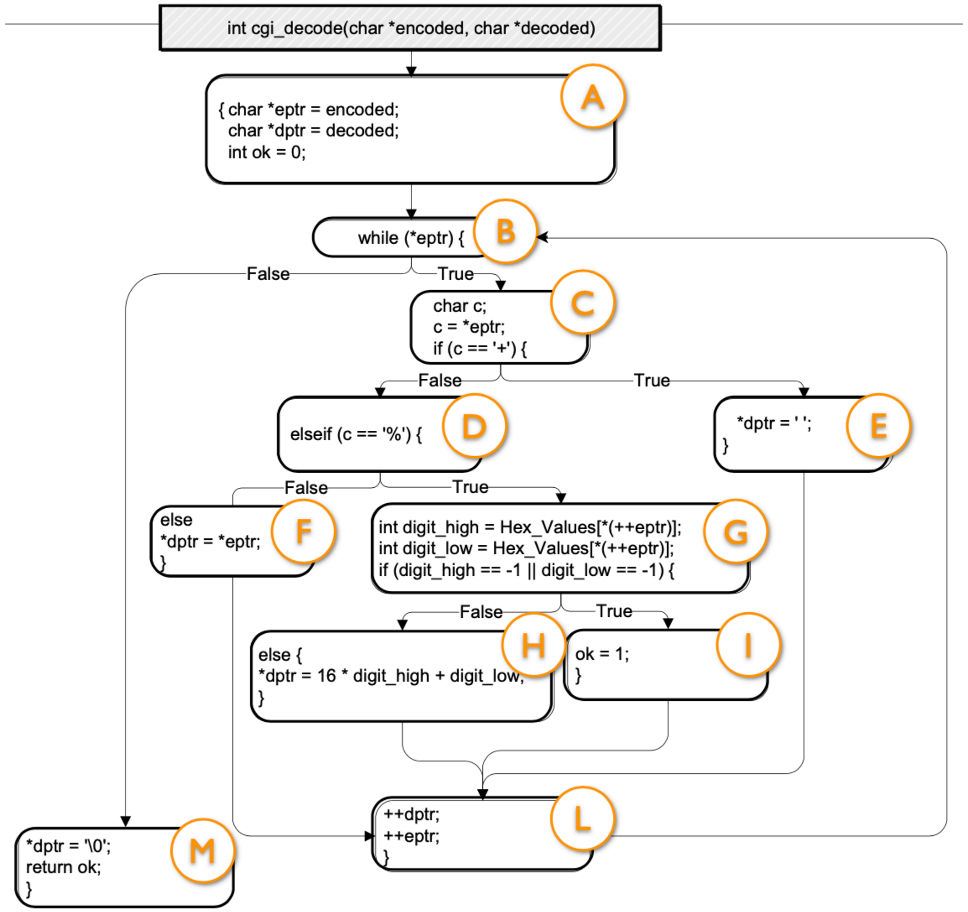
- Structural Testing is also called “white-box testing.”
- Purpose is to exercise code implementation.
- Adequacy can be measured as %ge of goal:
  - Statement coverage
  - Branch coverage
  - Path coverage
- Quantitative measurement is possible.

Structural Testing Example (1)

- Break function into basic blocks
- Build a Control-Flow Graph (CFG)

```
int cgi_decode(char *encoded, char *decoded) {
  char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;
  while (*eptr) /* loop to end of string ('\0' character) */
  {
    char c;
    c = *eptr;
    if (c == '+') {  /* '+' maps to blank */
      *dptr = ' ';
    } else if (c == '%') { /* '%xx' is hex for char xx */
      int digit_high = Hex_Values[*(++eptr)];
      int digit_low  = Hex_Values[*(++eptr)];
      if (digit_high == -1 || digit_low == -1)
        ok = 1; /* Bad return code */
      else
        *dptr = 16 * digit_high + digit_low;
    } else { /* All other characters map to themselves */
      *dptr = *eptr;
    }
    ++dptr; ++eptr;
  }
  *dptr = '\0';  /* Null terminator for string */
  return ok;
}
```

Structural Testing Example (2)

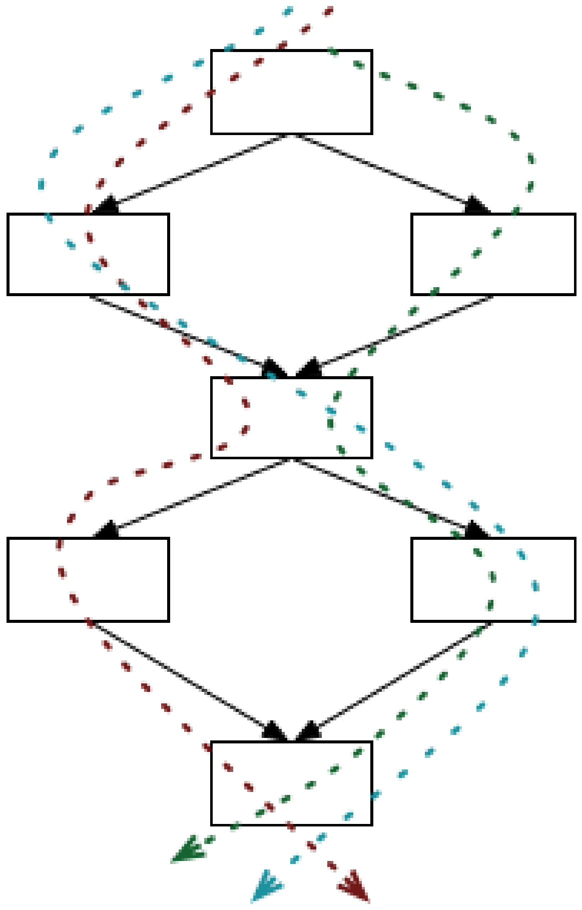


- Evaluating Tests:
  - “test” (A,B,C,D,F,L,M)
  - “a+b” (A,B,C,D,E,F,L,M)
  - “%3d” (A,B,C,D,G,H,L,M)
  - “%g” (A,B,C,D,G,I,L,M)
- Altogether, 100% *block* coverage
  - (first test could be omitted)
- Also 100% *branch* coverage
- If block “F” were absent, “%3d+%g” gets 100% block coverage while missing a branch.

(from Pezze + Young, “Software Testing and Analysis”, Chapter 12)

Structural Testing: Paths

- Sometimes a fault is only manifest on a particular path
  - E.g., choosing the left branch and then choosing the right branch. (dashed blue path)
- But the number of paths can be infinite
  - E.g., if there is a loop.
- There are ways to bound the number of paths to cover.



Structural Test Criteria

1. Path coverage (usually impossible) *implies*
2. Repetition-Free Path Coverage *implies*
3. Branch Coverage *implies*
4. Block Coverage = Statement coverage.

(Other coverage criteria exist, some incomparable)

See https://en.wikipedia.org/wiki/White-box_testing

100% Coverage may be Impossible

- Path coverage (even without loops)
  - Dependent conditions: `if (x) A; B; if (x) C;`
- Edge coverage
  - E.g., `if (x < 0) A; else if (x == 0) B; else if (x > 0) C;`
- Statement coverage
  - Dead code (e.g., defensive programming)

Mutation Testing

- Mutation testing is a form of structural testing
- The code in the SUT is mutated
  - E.g., replacing "&&" with "`||`" in an "`if`" statement.
- Then we see if the test suite fails.
- Mutation testing is more than coverage, because it checks that the change made a difference.
- Difficult in practice:
  - Too many mutants possible (time)
  - Too many mutants are equivalent or uninteresting:
    - `rpc.set_deadline(10);` ⟶ `rpc.set_deadline(20);`

But possible!
https://research.google/pubs/pub46584/

Adequacy of Regression Tests (1)

- Regression tests control maintenance:
  - A change cannot be committed until "all" tests pass.
    - Often "all tests" means "all *small automated unit* tests"
- Adequacy includes whether tests cover all *uses:*
  - Uses may include unspecified behavior:
    - E.g., Users may assume that a hash result is non-negative;
    - Hyrum's law: any visible behavior may have dependents.
- Users are responsible to add tests:
  - Beyoncé rule: "If you liked it you should have put a ~~ring~~ **test** on it" (SoftEng @ Google)

Adequacy of Regression Tests (2)

- *Flaky* tests are those that fail intermittently:
  - Nondeterminism (e.g., hash codes, random numbers);
  - Timing issues (e.g., threads, network).
- *Brittle* tests are those that fail when tests changed:
  - Ordering (e.g., assume prior state)
- *Mystery* tests aren’t clear why they fail:
  - How can the developer know what to do to fix?
- All these impede maintenance:
  - A capricious, rigid or incomprehensible gatekeeper impedes the ability to make progress.

These definitions are not universal.

Adequacy of Regression Tests (3)

- “Test Smells” name problem aspects of tests:
  - “Smell” = “Disagreeable Odor” (metaphor)
  - Can be seen when reviewing tests;
  - Named (as Design Patterns) for communication.
- Two lists of ”Test Smells”:
  - van Deursen et al. Refactoring test code
  - https://www.peruma.me/project/test-smells/
- Smelly tests more likely to be flaky, brittle, mysterious or otherwise “bad.”
- Some examples on next slides.

Test Smells (1)

```
it('writes right', () => {
  const w =
   fs.createWriteStream('test.txt');
  const t = createBigTree();
  t.write(w);
  w.end();
  const d =
    fs.readFileSync('test.txt');
  /* … check result … */
}
```

Resource Optimism

- Assumes that certain external resources can be used.

Problem:

- If assumption proves false, test becomes "flaky."
- Here we are assuming "test.txt" is writable and not being used by something else (e.g. this same test being run in parallel).

Test Smells (2)

```
it('remove only removes one', () =>{
  const tree = makeBST();
  for (let i = 0; i < 1000; ++i) {
    tree.add(i);
  }
  for (let j = 0; j < 1000; ++j) {
    for (let i = 0; i < 1000; ++i) {
      if (i != j) tree.remove(i);
    }
    expect(tree.contains(j)).
      toBe(true);
  }
}
```

CONDITIONAL TEST LOGIC

- Test code has conditionals/loops

Problem:

- Test is hard to understand.
- If it fails, no clue as to what went wrong:
  - "false is not true"
- Test is a "mystery" test.

(Incidentally, also suffers from hard-coding 1000 in the test.)

Test Smells (3)

```
it(‘removes max’, ()=>{
  tree.remove(31);
  expect(tree.size()).
    toBe(4);
}
```

MYSTERY GUEST

- Uses information unknown to test;
- Assumes (mutable) context.

Problem

- Test will mis-behave if reordered
- Test is “brittle.”

What Makes Tests Good

- Tests should be **hermetic**
  - Reduce flakiness.
- Tests should be **clear**
  - After failure, should be clear what went wrong.
- Tests should be scoped as **small** as possible
  - Faster and more reliable.
- Tests should make calls against **public** APIs
  - Or they become brittle.

For a fuller treatment:
https://learning.oreilly.com/library/view/software-engineering-at/9781492082781/ch12.html#unit_testing

Review

- Now that you've studied this lesson, you should be able to:
  - Describe measures of test suite adequacy, and to know their limitations;
  - Distinguish flaky and brittle tests;
  - Name several test smells, and give examples;
  - Explain some properties of good tests.

Looking ahead

- In the next lesson, we will look closer at system tests.