Test Strategies

Michael Hilton and **Rohan Padhye**

Administrativia

- HW4 Part B (microservice feature implementation) due tonight
- HW4 Part C is now due Nov 8 (next Monday)
- HW4 Part D is still due Nov 9 (next Tuesday) – don’t wait till N-1

Learning goals

- Enumerate various strategies for picking test cases, such as:
    - Specification-based testing
    - Boundary-value testing
    - Structural testing
    - Property testing
    - Regression testing
    - Differential testing
    - Property-based testing
    - Mutation testing
- Contrast the

How many ways can you test the sort() function?

Basic Unit Test

```
@Test
public void testSort() {
    var input : List<Integer> = Arrays.asList(1, 2, 3);
    var output : List<Integer> = Arrays.asList(1, 2, 3);
    assert sort(input).equals(output);
}
```

Is this test good enough?

Q1: What are some interesting values to test?

List tuples <input, output, reason>

Black-box & Specification-Based Testing

- Test cases are often designed based on behavioral equivalence classes.
  - *Assumption*: if test passes for one value => test will pass for all values in the equivalence class.
- Systematic tests can be drawn from specification.
  - For example: A year is a *leap year* if:
    - the year is divisible by 4;
    - and the year is not divisible by 100;
      - except when the year is divisible by 400
  - Tests:
    - assert isLeapYear(1945) == false
    - assert isLeapYear(1944) == true
    - assert isLeapYear(1900) == false
    - assert isLeapYear(2000) == true

Boundary-Value Testing

- *Aim*: Test for cases that are at the "boundary" of equivalence classes in the specification.
    - Small change in input moves it from one class to another.
    - Example: Testing a function *divide*(int a, int b)
        - One boundary may be at `a == b`

- *Edge case:* One of many parameters are at the boundary
    - E.g. for *divide*: a=0, b=42 or a=42, b = 0
    - E.g. for *sort*: list contains duplicates, list is empty

- *Corner* case: Combination of parameters are at the boundary
    - E.g. for *divide*: a=0, b=0

Boundary-Value Testing

- Let’s see some (non-systematic) examples for boundary testing sort().
- *Reflect*: Will these tests work well for all sort algorithms?

White-box or Structural Testing

- *Aim*: Test for cases that exercise various program elements (e.g. functions, lines, statements, branches)

- *Key idea*: If you don't execute some code, you can't find bugs in that code. So, let's execute all the code.

Coverage of the Basic Unit Test

```
@Test
public void testSort() {
    var input :List<Integer> = Arrays.asList(1, 2, 3);
    var output :List<Integer> = Arrays.asList(1, 2, 3);
    assert sort(input).equals(output);
}
```

Element	Class, %	Method, %	Line, %
BubbleSort	100% (1/1)	100% (1/1)	80% (8/10)
CocktailShakerSort	100% (1/1)	100% (1/1)	72% (13/18)
CombSort	100% (1/1)	100% (2/2)	92% (12/13)
CycleSort	100% (1/1)	50% (1/2)	34% (8/23)
GnomeSort	100% (1/1)	100% (1/1)	66% (6/9)
HeapSort	100% (2/2)	100% (5/5)	88% (40/45)
InsertionSort	100% (1/1)	100% (1/1)	77% (7/9)
MergeSort	100% (1/1)	100% (3/3)	91% (22/24)
PancakeSort	100% (1/1)	100% (2/2)	100% (14/14)
QuickSort	100% (1/1)	100% (4/4)	100% (23/23)
SelectionSort	100% (1/1)	50% (1/2)	57% (8/14)
ShellSort	100% (1/1)	100% (1/1)	83% (10/12)
SortAlgorithm	100% (1/1)	50% (1/2)	50% (1/2)
SortUtils	100% (1/1)	100% (3/3)	100% (6/6)
TimSort	100% (2/2)	29% (5/17)	5% (22/374)

Q2: Which one do you think is harder:
black-box boundary-value testing
or
white-box structural testing?

But the basic unit test worked well for QuickSort....

COVERAGE != COMPLETENESS

Mutation Testing

- *Key idea*: Inject bugs in the program by *mutating* the source code.
- *Ideally*: at least one test should fail on the mutated program (= catch bug).
  - If this happens, the mutant is said to be “killed”.
  - If all tests continue to pass under the mutated program, then the mutant is said to “survive”.
  - Mutation score = (mutants killed) / (total mutants). This is a better predictor of bug-finding capability than coverage.
- *Competent programmer assumption*: programs are mostly correct, except for very small errors.
  - Shows that tests are falsifiable at the boundary of implementation (as opposed to boundary of specification).

Mutation Testing

- Sample mutations include:
  - Change ‘a + b’ to ‘a – b’
  - Change ‘if (a > b)’ to ‘if (a >= b)’ or ‘if(b > a)’
  - Change ‘i++’ to ‘i—’
  - Replace integer variables with 0
  - Change ‘return x’ to ‘return True’ (or some other constant)
  - Delete lines containing void method calls (e.g. ‘x.setFoo(1)’)
  - … and many more
- Over time, standard list of mutators curated by researchers
- PIT is a popular mutation testing tool for Java (pitest.org)

Q3: Suggest some mutations to (this snippet from) QuickSort

```java
  private static <T extends Comparable<T>> int partition(T[] array, int left, int right) {
    int mid = (left + right) >>> 1;
    T pivot = array[mid];

    while (left <= right) {
      while (less(array[left], pivot)) {
        ++left;
      }
      while (less(pivot, array[right])) {
        --right;
      }
      if (left <= right) {
        swap(array, left, right);
        ++left;
        --right;
      }
    }
    return left;
  }
}
```

Mutation Testing

- Nice idea but has several limitations:
    1. *Equivalent* mutations: Modifications that do not affect program semantics (e.g. affecting the pivot in Quicksort).
    2. Needs a pretty *complete* test oracle: Otherwise, some genuine bugs may never be caught. We'll come back to this point later.
    3. Expensive to run. N mutants require N test executions. Program testing costs scale quadratically (because N also grows with size).

```java
private static <T extends Comparable<T>> int partition(T[] array, int left, int right) {
  int mid = (left + right) >>> 1;
  T pivot = array[mid];

  while (left <= right) {
    while (less(array[left], pivot)) {
      ++left;
    }
    while (less(pivot, array[right])) {
      --right;
    }
```

Carnegie Mellon University
School of Computer Science

Test Oracles

- Obvious in some applications (e.g. “sort()”) but more challenging in others (e.g. “encrypt()” or UI-based tests)

- Lack of good oracles can limit the scalability of testing. Easy to generate lots of input data, but not easy to validate if output (or other program behavior) is correct.

- Fortunately, we have some tricks.

Property-Based Testing

- Intends to validate invariants that are always true of a computed result.
  - E.g. if testing a list-reversing function called `rev`, then we have the invariant: `rev(rev(list)).equals(list)`

- Key idea: Can now easily scale testing to very large data sets, either hand-written or automatically generated, without the need for hard-coding expected outputs completely.

```java
@Property
public void testSameLength(List<Integer> input) {
    var output : List<Integer> = sort(input);
    // Check length
    assert output.size() == input.size() : "Length should match";
}
```

Q4: What are some other properties of that should be true of the result of sort()?

(is there a complete set?)

```
@Property
public void testSameLength(List<Integer> input) {
    var output : List<Integer> = sort(input);
    // Check length
    assert output.size() == input.size() : "Length should match";
}
```

Differential Testing

- If you have two implementations of the same specification, then their output should match on all inputs.
  - E.g. `timSort(x).equals(quickSort(x))` → should always be true
  - Special case of a property test, with a free oracle.

- If a differential test fails, at least one of the two implementations is wrong.
  - But which one?
  - If you have N > 2 implementations, run them all and compare. Majority wins (the odd one out is buggy).

- Differential testing works well when testing programs that implement standard specifications such as compilers, browsers, SQL engines, XML/JSON parsers, media players, etc.
  - Not feasible in general e.g. for CMU's custom grad application system.

Regression Testing

- Differential testing through time (or *versions*, say V1 and V2).
- Assuming V1 and V2 don't add a new feature or fix a known bug, then f(x) in V1 should give the same result as f(x) in V2.
- *Key Idea*: Assume the current version is correct. Run program on current version and log output. Compare all future versions to that output.

Takeaways

- Most tests that you will write will be muuuuuuch more complex than testing a sort function.
- Need to set up environment, create objects whose methods to test, create objects for test data, get all these into an interesting state, test multiple APIs with varying arguments, etc.
- Many tests will require mocks (i.e., faking a resource-intensive component).
- General principles of many of these strategies still apply:
  - Writing tests can be time consuming
  - Determining test adequacy can be hard (if not impossible)
  - Test oracles are not easy
  - Advanced test strategies have trade-offs (high costs with high returns)