CS1632:
Property-Based Testing

Wonsun Ahn

What is Testing?

- Checking *expected behavior* against *observed behavior*
- What we have been doing so far:
    1. Split the set of input values into equivalence classes
    2. Choose a few representative values from each equivalence class
    3. Write test case for those few values

    … And hope that those few values cover all behavior

- But do they? Are you really confident?

So let’s take a sort function

```
public int[] sort(int[] arrToSort) {
    ...
}
```

Possible test cases

- null
- []
- [1]
- [-1]
- [1, 2, 3, 4, 5]
- [5, 4, 3, 2, 1]
- [-9, 7, 2, 0, -14]
- [1, 1, 1, 1, 1, 1]
- [1, 2, 3, 4 … 99999, 100000, 100001]

At what point would you be satisfied?

- There is a near infinite number of sequences you could test
- Each sequence is a unique equivalence class! (almost)
  - In the sense that each sequence will take a different execution path
  - Each execution path represents a unique behavior
- Verdict: it’s impossible to write enough tests to cover all behavior

- Well.. It’s impossible for a human, but can’t we auto-generate them?

Stochastic Testing

- Stochastic testing: testing using randomly generated input values
  - Note: we are still not testing all input values
  - We are just testing a large number of random values hoping good coverage
- Popularly called “monkey testing” (monkey on the typewriter)
  - Not a good analogy: implies no thought is given to generation of input values
  - Testers should give *a lot* of thought to how input values are generated
  - Values are generated from a distribution, and distribution affects coverage
  - Testers should choose a distribution most likely to uncover defects

Good Distribution of Values is Important

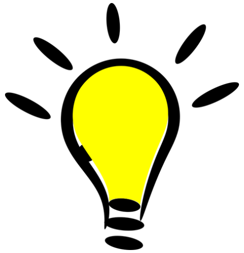
- For example, suppose we have a defective method foo:

```
void foo (int x, int y) {
  if (x == y) {
    // Defect occurs
  }
}
```

- If we blindly test random combinations of `x` and `y`
  → Low probability that combination where `x == y` is chosen
  → Low probability that defect will be detected even with randomization
- A good distribution will contain cases where `x == y`

The Test Oracle Problem

- So now we have a set of randomly generated input values
- Now how do we auto-generate the expected output values?
  - Using the tested code? No, that is observed, not expected output value.
  - We need an “oracle” to tell us what the expected outcome should be.
  - Otherwise, output values must be calculated by humans one by one.
  - This is called the *Test Oracle Problem.*



What if we tested ***properties*** instead of output values?

Property-Based Testing

- *Property-Based Testing*: testing correctness properties of observed values
  - *Property*: something that must invariably hold true in observed values
  - Properties are also called *invariants*
  - Does not test "observed values == expected values"

  → No need to generate expected values as part of the test!

Properties allows testing of any random input value.

Examples of Invariants

- Invariants on statement `a = b + c;`

  ```
  Assuming b > 0 && c > 0, a > 0 always holds true
  Assuming b < 0 && c < 0, a < 0 always holds true
  ```

- Invariants on statement `a = b - c;`

  ```
  Assuming b > c, a > 0 always holds true
  Assuming b < c, a < 0 always holds true
  ```

- Property-based testing does not always guarantee correctness
  - But if you check enough properties, you can often get pretty close

Going back to our sort() example

```
public int[] sort(int[] arrToSort) {
    ...
}
```

- What are the invariants?

1. Output array is the same size as input array
2. Every element in input array is in output array
3. No element not in input array is in output array
4. Values in output array are always increasing or staying the same
5. Idempotent: `sort(arr) == sort(sort(arr))`

- If all these invariants hold, by definition, the sort is provably correct!

Property-Based Testing Pros / Cons

- Disadvantages
  - Does not always guarantee that each output value will be correct
- Advantages
  - Can check behavior without being provided expected output values
    → Enables stochastic testing
    → Can be embedded in source code to check values at runtime
    (`java -ea Foo` enables all `assert`s in Foo due to `-ea` flag)
  - Leads coder to think about invariants and improve code. E.g.

```
int[] result = sort(arr);
if ( result.length != arr.length ) {
  // Handling of the case where lengths differ
}
```

    → Invariant is `result.length == arr.length`, so why the `if`?

Stochastic Testing Pros / Cons

- Advantages
  - Can achieve high test coverage with minimum effort
  - Test input values not biased by preconceptions tester has about code
  - Coders get to understand invariants in code, which leads to code improvement

- Disadvantages
  - Due to property testing, does not always guarantee output value correctness
  - Tests are **not repeatable**

Stochastic Testing and Repeatability

- Reproducibility vs. Repeatability
  - Reproducibility: Ability to reproduce a defect found by a test
  - Repeatability: Ability of a test to repeat itself every time it is run
- Stochastic tests **are reproducible**
  - Tester only needs to record randomly generated input value causing defect
  - Tester can pass on input value to coder to reproduce the defect
- Stochastic tests **are not repeatable**
  - Every time the test runs, a new input value is tested by design
  - Even when defect is present, test may pass or fail depending on time of day
  - Very hard to figure out when the defect was introduced
    (Where should we revert to? Previous version, 10 versions ago, 100 versions ago?)

Stochastic Testing Best Practice

- Stochastic testing should not be your main method of testing

- Use repeatable tests to the fullest --- tests that use concrete values
  - Test all the edge cases and corner cases that are frequent sources of defects
  - This should be your first line of defense

- Use stochastic testing as a second line of defense
  - To catch unexpected defects that testers didn’t even think to test
  - Hard to deal with due to non-repeatability, but better than not catching defect

QuickCheck: Tool for Stochastic Testing

- Presented at ICFP ‘00 in the paper, “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs” https://www.researchgate.net/publication/2449938_QuickCheck_A_Lightweight_Tool_for_Random_Testing_of_Haskell_Programs
- More popular in functional programming languages
  - Because all functions in functional programming are pure by definition
  - Pure functions are easier to use property-based testing on (Don’t need to reason about side-effects when writing properties)
- But becoming more mainstream in other languages too

Not just used in functional programming!

- Java: junit-quickcheck
- Ruby: rantly
- Scala: scalacheck
- Python: pytest-quickcheck
- Node.js: node-quickcheck
- PHP: php-quickcheck
- Clojure: simple-check
- C++: QuickCheck++
- .NET: FsCheck
- Erlang: Erlang/QuickCheck

Using QuickCheck

- Two simple steps:
  1. Specify the properties of the allowed input
  2. Specify the properties of the output that must hold (invariants)

Example junit-quickcheck tests

```
@Property public void testConcat(String s1, String s2) {
  assertEquals(s1.length() + s2.length(),
               (s1 + s2).length());
}
@Property public void testSqrt(@InRange(minInt=0) int n)
{
  if(n > 1) assertTrue(n > sqrt(n));
}
```

- `@Property`: 100 randomized trials are done on the property-based test
- `@InRange`: constrains the range of randomized input values

Write the junit-quickcheck test for sort()

```
@Property public void testSort(int[] arr) {
  int[] result = sort(arr);
  assertEquals(arr.length, result.length);
  …
}
```

Then sit back with a beverage of your choice

- QuickCheck then runs randomized test cases for us!

COMPUTER – DOING HARD WORK!

```
[17, 19, 1] -> [1, 17, 19] OK
[-9, -100] -> [-100, -9] OK
[8, 2, 987, 287, 201] -> [2, 8, 201, 287, 987] OK
[101, 20, 32, -4] -> [-4, 20, 32, 101] OK
[115] -> [115] OK
[2, -9, -9, 1, 2] -> [-9, -9, 1, 2, 2] OK
[8, 3, 0, 4] -> [0, 3, 4, 8] OK
[17, 1009, -2, 413] -> [-2, 17, 413, 1009] OK
[12, 12, 1, 17, -100] -> [-100, 1, 12, 12, 17] OK
[] -> [] OK
...
```

YOU –
lying on
beach
taking
foot selfies!

This is what it sounds like when Invariants fail

```
[17, 19, 1] -> [1, 17, 19] OK
[-9, -100] -> [-100, -9] OK
[8, 2, 987, 287, 201] -> [2, 8, 201, 287, 987] OK
[101, 20, 32, -4] -> [-4, 20, 32, 101] OK
[115] -> [115] OK
[2, -9, -9, 1, 2] -> [-9, -9, 1, 2, 2] OK
[8, 3, 0, 4] -> [0, 3, 4, 8] OK
[17, 1009, -2, 413] -> [-2, 17, 413, 1009] OK
[12, 12, 1, 17, -100] -> [-100, 1, 12, 12, 17] OK
[9, 0, -6, -5, 14] -> [0, -6, -5, 9, 14] FAIL
[] -> [] OK
```

Shrinking

```
[9, 0, -6, -5, 14] -> [0, -6, -5, 9, 14] FAIL
[9, 0, -6] -> [0, -6, 9] FAIL
[-6, -5, 14] -> [-6, -5, 14] OK
[9, 0] -> [0, 9] OK
[0, -6] -> [0, -6] FAIL
[0] -> [0] OK
[-6] -> [-6] OK

Shrunk Failure: [0, -6] -> [0, -6]
```

Shrinking

- Finds the smallest possible input that triggers a failure
- Helps track down actual issue
- A “toy” failure is a great thing to add to a defect report

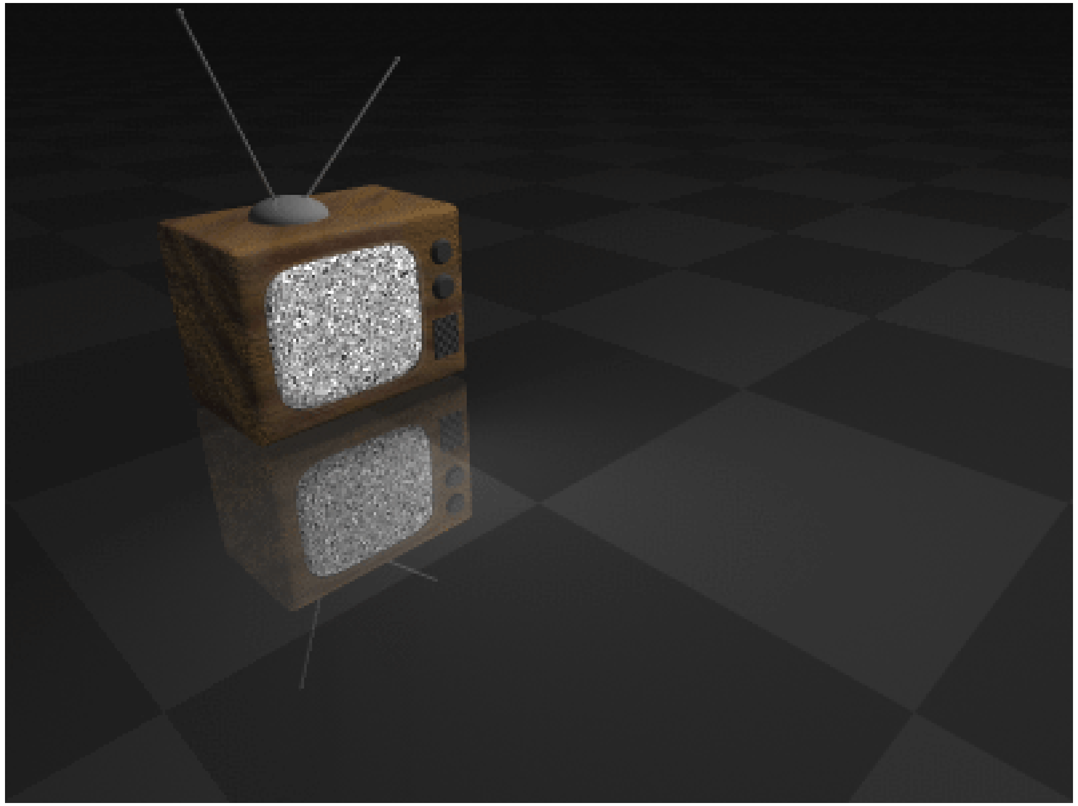
Adjusting Trials per Run

```
@Property(trials=1000) public void testSort(int[] arr) {
  int[] result = sort(arr);
  assertEquals(arr.length, result.length);
  …
}
```

- By default, 100 randomly generated input values are passed to test method
- You can adjust the number of trials to `1000` using the `trials` attribute

Fuzz Testing

Fuzz (*noun*): a blurred effect



Fuzz Testing

- A form of stochastic testing with a focus on “byte stream” inputs
- Idea: feed program with “fuzzy” or “noisy” byte streams and see if it
  - Exposes a defect (most commonly a system crash)
  - Exposes a security vulnerability

- A byte stream can be as varied as …
  - An input file to an image viewer or video player
  - A network packet to a web server
  - JavaScript code interpreted by a web browser
  - A configuration file for a program

Fuzz Testing is not Dumb Testing

- Fuzz testing may sound like stupid testing
  - Generate some random static and force feed it to your program until it fails!
- But you will never be effective this way
  - In fact, you have to be even more careful how you generate inputs
  - A byte stream is a complex form of input that leads to complex behavior
  - Unlike a simple integer input where all behavior can be covered relatively easily

Why completely random input is ineffective

- Suppose we are testing a web browser
- Here is our test plan:
    1. Generate a set of randomized strings and store them into HTML files
    2. See if any of the HTML files crashes the web browser
- Is this the best way to test the robustness of our browser? Why (not)?
- I vote for NO
    - A browser starts by first checking the integrity of the HTML file
      (e.g. has necessary tags such as <html>, is structured correctly, etc …)
    - 99.99% of the randomized files will fail the initial check
    - 99.99% of the randomized files will achieve very poor code coverage
- We need a way to generate inputs with minimal integrity and structure

A Smarter Algorithm for Fuzz Testing

- New test plan for web browser
  1. Start from a collection of existing HTML files (call it the *corpus*)
  2. "Fuzz" HTML files in *corpus* to create new variants and add to *corpus*
- Steps to "fuzz" HTML files
  1. Parse an HTML file
  2. Mutate parts of parse tree with new values
     - Optionally from a dictionary; dictionary contains HTML tag names etc.
  3. Regenerate HTML file from parse tree and test on web browser
  4. Only add to *corpus* if the new HTML file increases code coverage
  5. Stop if sufficient code coverage is achieved, otherwise loop back to 1.
- Ensures HTML files have minimal integrity and structure

Now Please Read Textbook Chapter 18

- Other References:

- “Fuzzing with Code Fragments” (USENIX Security 2012)
  - https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final73.pdf
  - Fuzz testing found 105 security vulnerabilities on Firefox JavaScript engine

- libFuzzer: an LLVM compiler library for coverage-guided fuzz testing
  - https://llvm.org/docs/LibFuzzer.html
  - Still experimental but can be used with any language handled by clang (`clang -fsanitize=fuzzer your_app.c`)