PROPERTY-BASED TESTING *(OCCASIONALY ILLUSTRATED WITH SELFIES)*

BILL LABOON

@BILLLABOON

LABOON@CS.PITT.EDU

WHAT IS TESTING?

By Jacques-Louis David - http://www.metmuseum.org/collection/the-collection-online/search/436105, Public Domain, https://commons.wikimedia.org/w/index.php?curid=28552

CHECKING

EXPECTED BEHAVIOR, B

AGAINST

OBSERVED BEHAVIOR, B'

UNDER

CIRCUMSTANCES, C

ASSUME AN ASCENDING SORT FUNCTION

```
// What values would you use to test this?
public int[] billSort(int[] arrToSort) {
    ...
}
```

POSSIBLE TEST CASES

- null -> null
- [] -> []
- [1] -> [1]
- [-1] -> [-1]
- [1, 2, 3, 4, 5] -> [1, 2, 3, 4, 5]
- [5, 4, 3, 2, 1] -> [1, 2, 3, 4, 5]
- [-9, 7, 2, 0, -14] -> [14, —9, 0, 2, 7]
- [1, 1, 1, 1, 1, 1] -> [1, 1, 1, 1, 1, 1]
- [1, 2, 3, 4 … 99999, 100000, 100001] -> [1, 2, 3, 4 … 99999, 100000, 100001]
- [4, 999, 19 … 22, 88765, 2345] -> [1, 2, 3, 4 … 99999, 100000, 100001]

LOTS OF TESTS TO WRITE!

- What if you forget an edge case?
- What if the test only works with the certain values you pass in?
- Lots of time will be spent writing test boilerplate.

WHAT OTHER EXPECTED BEHAVIOR COULD WE CHECK BESIDES THE SPECIFIC VALUE BEING RETURNED FROM THE METHOD?

PROPERTIES!

- First determine:
  - the properties of values to be passed in
  - the expected properties of the return value, given that input
- Then let the computer come up with test cases for us!

EXPECTED PROPERTIES VS OBSERVED PROPERTIES

- Note that properties are a subset of “behavior”!
- In traditional testing, our expected behavior would be listed as specific expected values…
    - … but this is not necessary to meet our definition of “testing”
    - … even if it is what we commonly think of as “testing”

A NEW KIND OF TESTING

- Presented at ICFP 2001 in the paper, “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”
  - http://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/quick.pdf
- Popular in functional programming world, and used in hardware testing, but becoming more mainstream in “traditional” software testing
- A subset of stochastic testing (using randomly or pseudorandomly generated values for testing)

REAL-LIFE EXAMPLE

- If I ask you to get a jar of peanut butter from a store, I want:
  - An item that exists in a store
  - Item contained in a jar
  - Contains peanuts and salt
  - Contains not more than 2% of anything other than peanuts and salt
  - Should have the term “peanut butter” somewhere on the label

NOW, SEND 100 PEOPLE TO THE STORE…

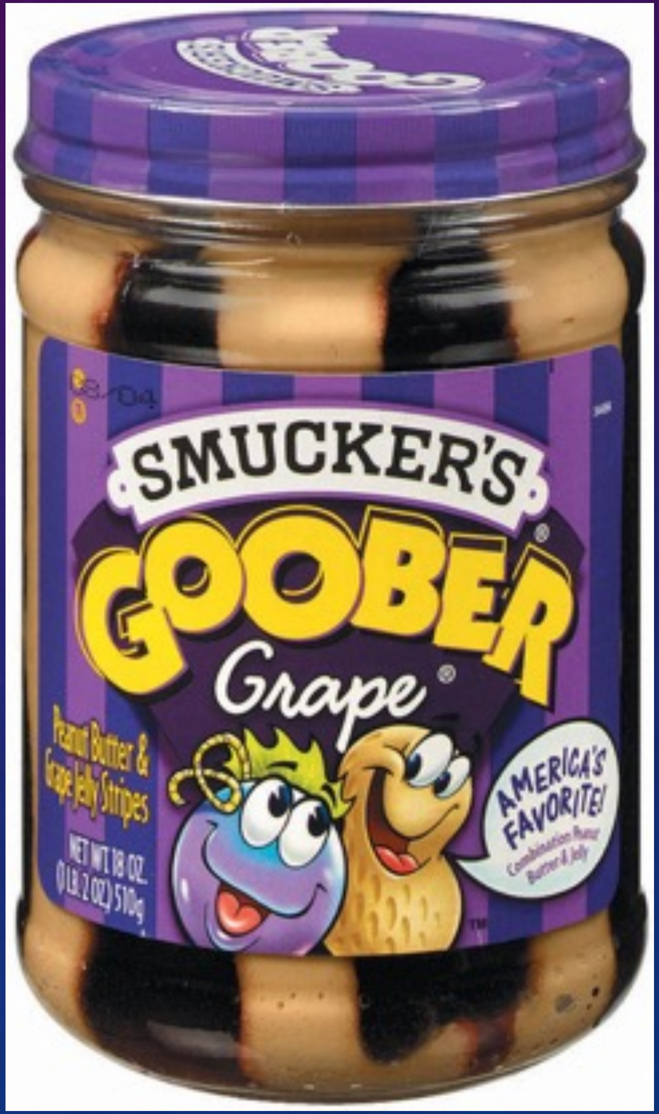
Peanut Butter & Co
The Heat Is On
peanut butter blended with fiery spices
NO TRANS FATS · NO CHOLESTEROL

FAILURE!

- An item that exists in a store
- ~~Item contained in a jar~~
- Contains peanuts and salt
- ~~Contains not more than 2% of anything other than peanuts and salt~~
- Should have the term "peanut butter" somewhere on the label

FAILURE!

- An item that exists in the store
- Item contained in a jar
- Contains peanuts and salt
- ~~Contains not more than 2% of anything other than peanuts and salt~~
- Should have the term “peanut butter” somewhere on the label


SMUCKER'S
GOOBER
Grape
Peanut Butter & Grape Jelly Stripes
NET WT 18 OZ (1 LB 2 OZ) 510g
AMERICA'S FAVORITE!
Combination Peanut Butter & Jelly

FAILURE!

- An item that exists in the store
- Item contained in a jar
- ~~Contains peanuts and salt~~
- ~~Contains not more than 2% of anything other than peanuts and salt~~
- ~~Should have the term “peanut butter” somewhere on the label~~

TWO STEPS

1. Specify the properties of the allowed input
2. Specify the properties of the output that should always hold
    - These properties are called *invariants*.

NO MORE PEANUT BUTTER, BACK TO SORTING

- What invariants would we expect of the input and output of a sorted array compared to the array passed in as an argument, assuming the following method signature?

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

INPUT PROPERTIES

1. Input will be either null or a reference to an array
2. If array is not null, will always contain 32-bit signed int values (between -2147483648 and 2147483647)
3. If array is not null, size will always be non-negative (0 or greater)

OUTPUT PROPERTIES

1. Output array same size as passed-in array
2. Values in output array always increasing or staying the same
3. Value in output array never decreasing
4. Every element in input array is in output array
5. No element not in input array is in output array
6. Idempotent - running it again should not change output array
7. Running it twice on same input array should always result in same output array

LET THE COMPUTER DO THE WORK

Now that we have the properties of expected input values, and the properties of expected output values, we can let the computer do the grunt work of developing specific tests. This is called property-based testing.

```
[0] -> [0]
[1, 3, 2] -> [1, 2, 3]
[-1, 19, 17, -22] -> [-22, -1, 17, 19]
```

THEN SIT BACK WITH A BEVERAGE OF YOUR CHOICE

- Based on our specifications, the computer then makes and runs our test suite 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!

FALSIFYING AN INVARIANT

```
[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 failure
- Helps track down actual issue
- A simplified, “toy” failure is always a great thing to add to a defect report

GENERATORS

- Input may be more complex than just “list of random ints”
- Example: User object with various attributes (ID, Username, Firstname, Lastname, etc.)
- *Generators* allow you to specify rules for what kinds of input will be generated
- Examples:
  - For an absolute value function, only pass in negative numbers for a test
  - Generating complex objects (e.g. various User objects with some values null, some Strings containing strange Unicode, some containing invalid data, etc.)
  - Generate input with more complex restrictions (e.g. only prime numbers)

NOT JUST FOR METHODS AND UNIT TESTING!

- System Testing Level
  - Check if random input to system causes issues
  - Check if nonexistent user names when logging in get any screen other than “Invalid user”
  - Check if users posting a message can cause issues
- Fuzzing (fuzz testing): pass in generated values, see if system crashes
  - Kind of property-based testing with one invariant, “system keeps running normally”

MORE USEFUL FOR..

- Mathematical functions
- Pure functions
- Well-specified problems
- Anything where a variety of inputs map to specific kinds of output

LESS USEFUL FOR…

1. Writing to a file
2. Communicating over a network
3. Displaying text or graphics
4. Subjective testing (e.g. user acceptance)
5. Impure functions in general

GENERAL BENEFITS

- Easy to run hundreds, thousands, or even millions of tests
- Can do so quickly, if tests are properly specified
- Avoids human bias
  - Humans may inadvertently only select "happy path" tests or ignore specific edge cases
- Works very well *if* invariants are specified correctly!

GENERAL DRAWBACKS

- It takes time and effort to correctly specify properties of input and output
  - If not done correctly, may cause false positives or miss defects!
- Due to reliance on randomness, may only find defects on *some* runs
  - Intermittent, non-deterministic failures!
- Human bias is sometimes a good thing!
  - A good tester may know where to productively focus their effort

PROPERTY-BASED TESTING IS *PART* OF A COMPLETE ~~BREAKFAST~~ TESTING STRATEGY

PROPERTY-BASED TESTING IS PROBABLY AVAILABLE IN YOUR LANGUAGE OF CHOICE

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

QUESTIONS?

PghQA talk (this talk) repository:
https://github.com/laboon/PGHQA-Talk

QuickCheck with Rust
(my talk at Rust Belt Rust 2016, more Rust-focused):
https://github.com/laboon/RBR_QuickCheck

Property-Based Testing with Ruby on Rails & Rantly
(code for Rails app with tests):
https://github.com/laboon/rantly_rails

Twitter: @BillLaboon
Email: laboon@cs.pitt.edu