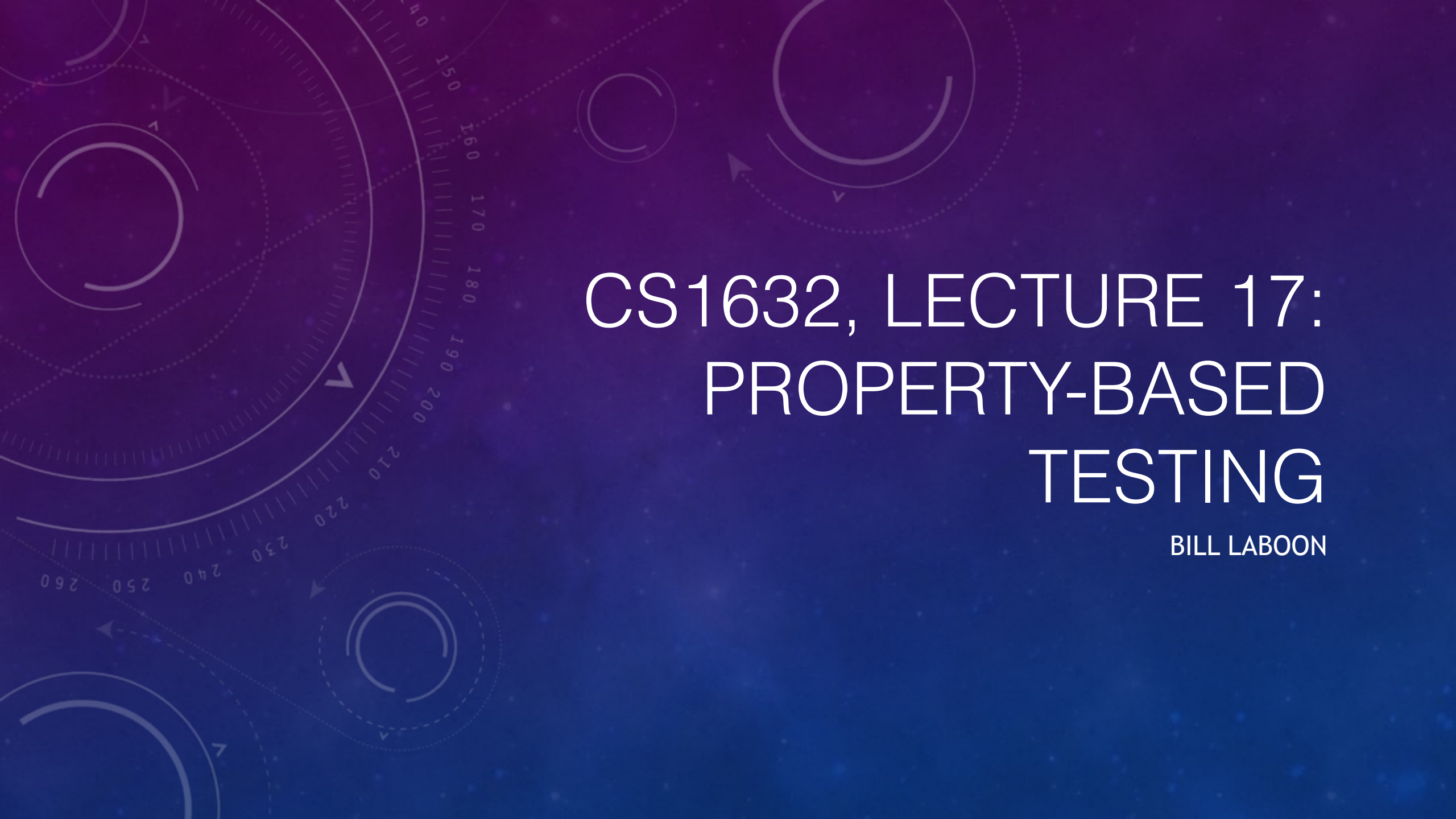

CS1632, LECTURE 17: PROPERTY-BASED TESTING

BILL LABOON

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

AGAINST

OBSERVED BEHAVIOR

OK, SO LET'S ASSUME A STANDARD SORT FUNCTION

```
public int[] billSort(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]

LOTS OF TESTS TO WRITE!

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

WHAT OTHER EXPECTED BEHAVIOR COULD WE CHECK BESIDES THE CORRECT VALUE BEING RETURNED?

PROPERTIES

- Let’s spell out:
  - the properties of what could 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 is a subset of “behavior”!
- Before, our expected properties were all “specific values”
  - but this is not necessary to meet our definition of “testing”

EXAMPLE

- What properties would we expect of the output of a sorted array compared to the array passed in as an argument?

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. Pure - 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]
```

A NEW KIND OF TESTING

- Presented at ICFP 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
- More popular in functional programming world (for various reasons) but becoming more mainstream

NOT JUST USED IN FUNCTIONAL PROGRAMMING!

- Java: junit-quickcheck
- Ruby: rantly
- Scala: scalacheck
- Python: pytest-quickcheck
- Node.js: node-quickcheck
- Clojure: simple-check
- C++: QuickCheck++
- .NET: FsCheck
- Erlang: Erlang/QuickCheck
- *The only one I couldn't find is a version for PHP.*

LESS USEFUL FOR…

1. Writing to a file
2. Communicating over a network
3. Displaying text or graphics
4. Impure functions in general

MORE USEFUL FOR..

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

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*.

THEN SIT BACK WITH A BEVERAGE OF YOUR CHOICE

- Based on our specifications, QuickCheck 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!

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 failure
- Helps track down actual issue
- A “toy” failure is a great thing to add to a defect report

THINK ABOUT THE LEVELS OF ABSTRACTION WE'VE JUMPED UP SINCE THE BEGINNING OF THE SEMESTER

1. Write and execute tests (manual testing)
2. Write tests, let computer execute
3. Write what KINDS of tests we want, let computer write tests and execute
   - With shrinking, will even try to track down the problem!