Property-based testing

17-313: Foundations of Software Engineering

https://cmu-313.github.io

Michael Hilton and **Josh Sunshine**

Spring 2026

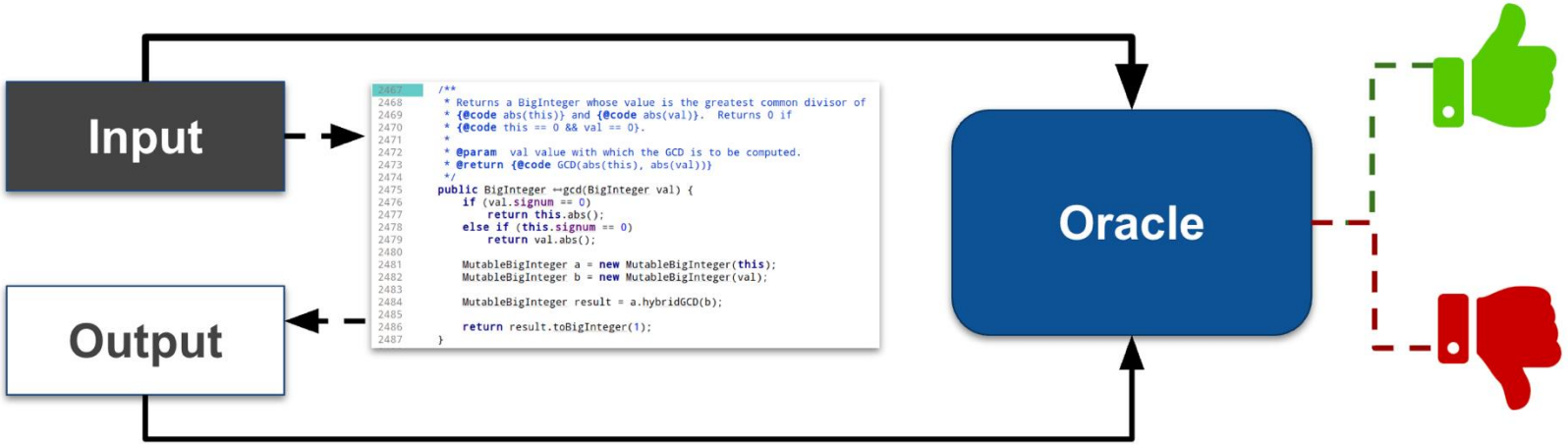
Learning Goals

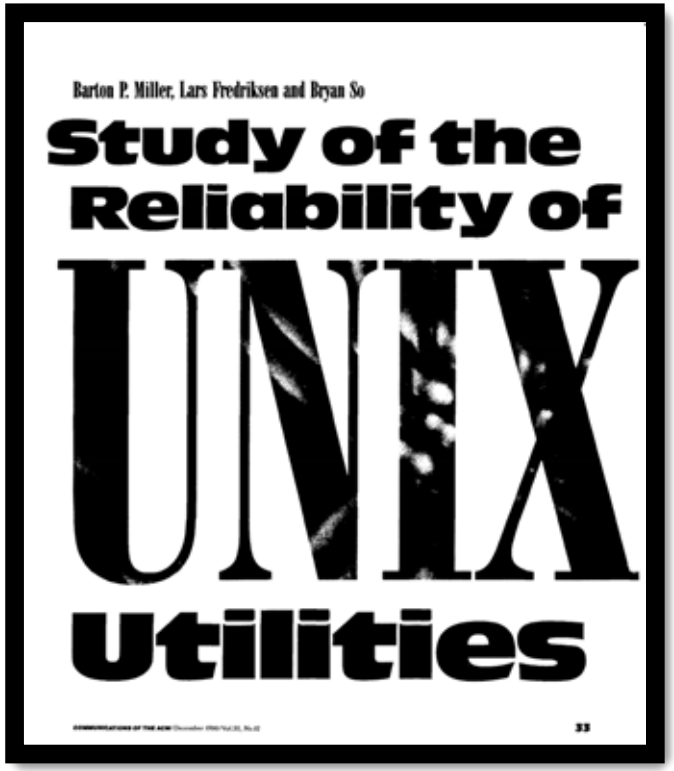
- Understand the tradeoffs between property-based testing to example-based testing
- Describe techniques for automatically generating input
- Identify common properties
- Analyze the quality of a test suite

Pre-break review: What if we automatically generate inputs? 

Problem: We need an Oracle!

- An oracle decides if behavior is correct for a given input
  - **strong oracles** catch bugs that **weak oracles** miss
  - designing strong oracles is difficult and often the bottleneck





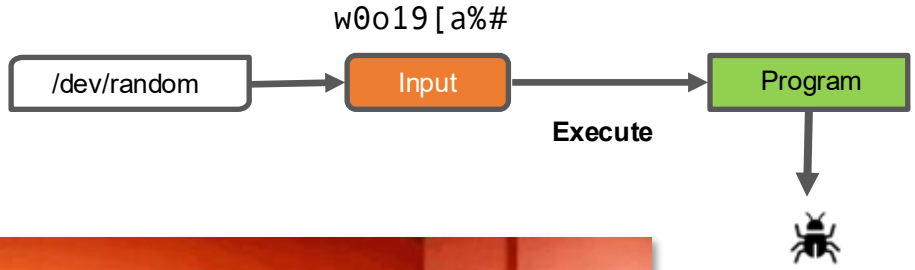
> On a dark and stormy night one of the authors was logged on to his workstation on a dial-up line from home and the rain had affected the phone lines; there were frequent spurious characters on the line. The author had to race to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising; but we were surprised that these spurious characters were causing programs to crash.

How can we identify these bugs?

Oracle: The Program Shouldn't Crash!

- **This is the oracle used by most fuzzing approaches**
- This oracle is a **generic property** that is not tied to any test inputs
  - that allows us to automatically generate and test any input
  - but the oracle is weak (i.e., not crashing does not imply correct)
- We can make the oracle slightly stronger by using **sanitizers**
  - detects illegal program states that might not cause an immediate crash
  - instruments the program at compile time (e.g., `-fsanitize=address`)
  - finds more safety issues but slows down execution / fuzzing
  - doesn't reveal logic bugs

Fuzz Testing randomly generates inputs and checks for program crashes



A 1990 study found crashes in:
adb, as, bc, cb, col, diction, emacs, eqn, ftp, indent, lex, look, m4, make, nroff, plot, prolog, ptx, refer!, spell, style, tsort, uniq, vgrind, vi

Oracle: Assertions in Source Code

- Assertions are **executable specifications**
  - document intended behavior (pre/postconditions, invariants)
- This oracle is generic and **not tied to any test inputs**
  - if we add assertions, we can use fuzzing to find some logic bugs!

```
function toUSD(amountCents: number): string {
  assert(Number.isInteger(amountCents), 'amount must be integer cents');
  assert(amountCents >= 0, 'amount must be non-negative');
  const dollars = (amountCents / 100).toFixed(2);
  return `$${dollars}`;
}
```

Property-based testing to the Rescue?

What is property-based testing?

- Introduced by the Haskell Library QuickCheck in 2000
  - Properties are “testable specifications” written in code
  - Advertised as “lightweight formal methods”
- Properties like assertions are automated oracles
  - Properties in testing code, assertions are in program code
- PBT in other languages:
  - Hypothesis for Python
  - **fast-check for JavaScript**

Property-Based Testing replaces single examples with properties that hold for many inputs

```
function isNonDecreasing(values: number[]): boolean {
  for (let i = 1; i < values.length; i++) {
    if (values[i - 1] > values[i]) return false;
  }
  return true;
}

fc.assert(fc.property(fc.array(fc.integer()), xs => {
  const ys = doubleBogosort(xs);
  return isNonDecreasing(ys);
}));
```

We want to test our implementation of an amazing sorting algorithm

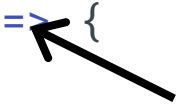
Property-Based Testing replaces single examples with properties that hold for many inputs

```
function isNonDecreasing(values: number[]): boolean {
  for (let i = 1; i < values.length; i++) {
    if (values[i - 1] > values[i]) return false;
  }
  return true;
}


fc.assert(fc.property(fc.array(fc.integer()), xs => {
  const ys = doubleBogosort(xs);
  return isNonDecreasing(ys);
}));
```

We use a generator to automatically explore inputs to our function under test

Property-Based Testing replaces single examples with properties that hold for many inputs

```
function isNonDecreasing(values: number[]): boolean {
  for (let i = 1; i < values.length; i++) {
    if (values[i - 1] > values[i]) return false;
  }
  return true;
}


fc.assert(fc.property(fc.array(fc.integer()), xs => {
  const ys = doubleBogosort(xs);
  return isNonDecreasing(ys);
}));
```



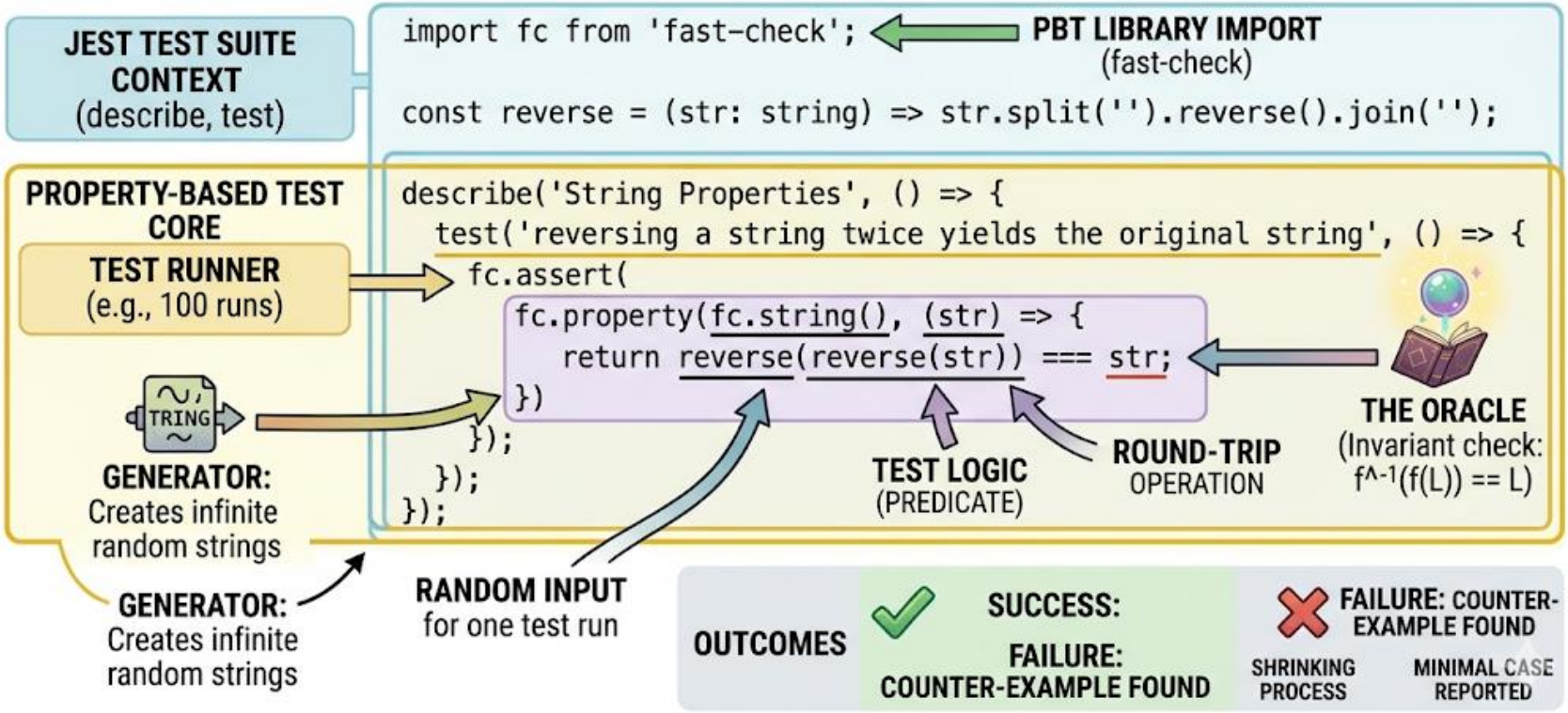
We provide a function that checks that a property holds under a given test input, xs

Property-Based Testing replaces single examples with properties that hold for many inputs

```
function isNonDecreasing(values: number[]): boolean {
  for (let i = 1; i < values.length; i++) {
    if (values[i - 1] > values[i]) return false;
  }
  return true;
}

fc.assert(fc.property(fc.array(fc.integer()), xs => {
  const ys = doubleBogosort(xs);
  return isNonDecreasing(ys);
}));
```

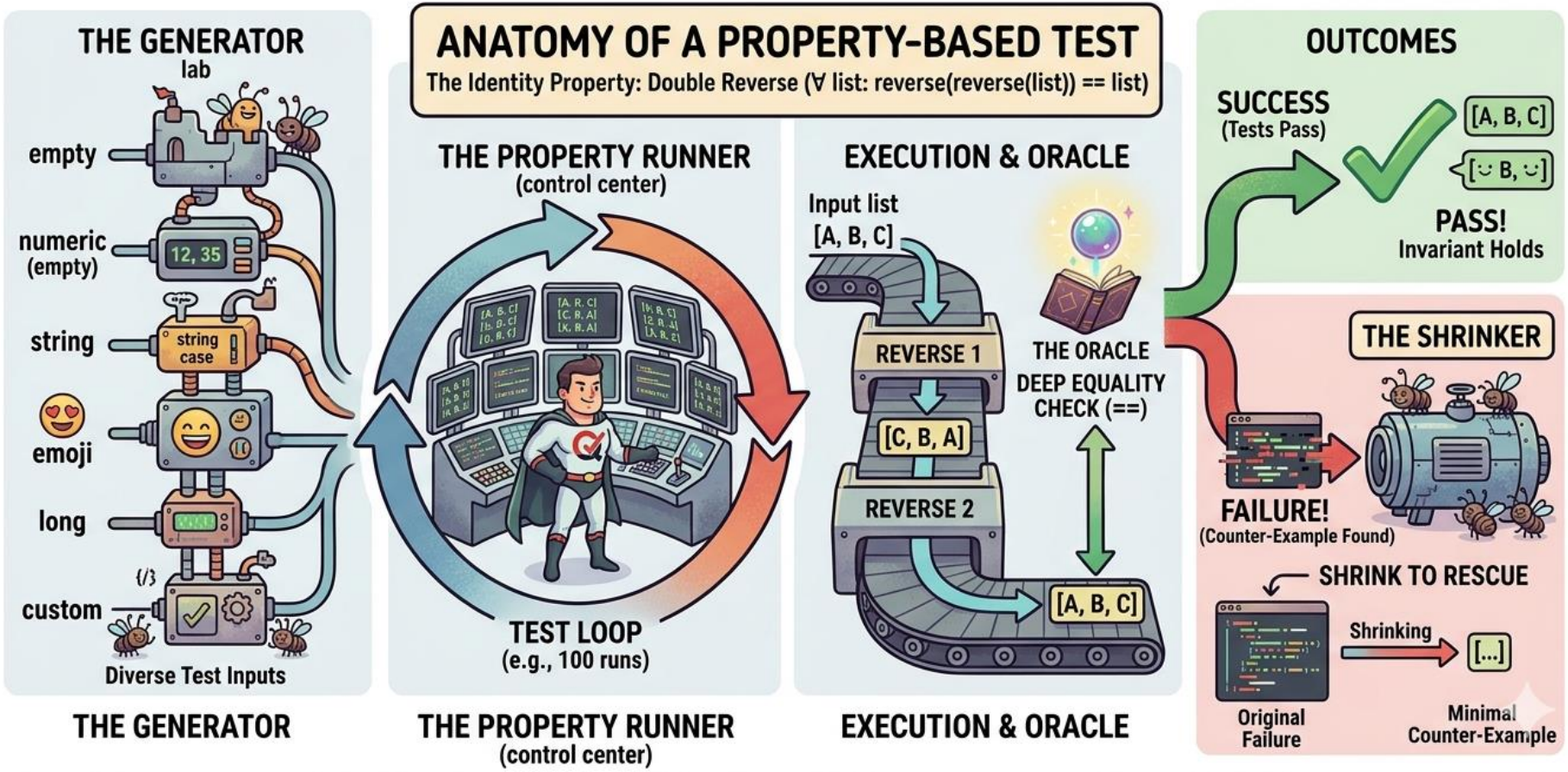
In this case, our property checks that array elements are non-decreasing

Anatomy of a Property-based test



ANATOMY OF A PROPERTY-BASED TEST

The Identity Property: Double Reverse (∀ list: reverse(reverse(list)) == list)



THE GENERATOR

THE PROPERTY RUNNER
(control center)

EXECUTION & ORACLE

Common Properties

Common Property: Round Tripping

- If you transform data and then undo it, you should get the original back. Mathematical logic: $f^{-1}(f(x)) = x$

```
fc.assert(
  fc.property(fc.string(), (str) => {
    return reverse(reverse(str)) === str;
  })
);

fc.assert(
  fc.property(fc.string(), (original) => {
    return decode(encode(original)) === original;
  })
);
```

Other common mathematical properties

- **Commutativity -** The order of inputs shouldn't matter for the result. Mathematical logic: $f(a, b) = f(b, a)$
- **Idempotence -** Running the operation multiple times has the same effect as running it once. Mathematical logic: $f(f(x)) = f(x)$
- **Monotonicity -** If the input increases (or stays the same), the output must also increase (or stay the same).
  Mathematical logic: *if* $x \leq y$ *then* $f(x) \leq f(y)$
- **Symmetry -** If Alice is the same as Bob, Bob is the same as Alice.
  Mathematical logic: *if* $a = b$ *then* $b = a$

In-class Exercise Part 1: Matching Game

Common Properties: Invariants

- Something **stays the same** or **within allowed bounds**
  - Sorting: same multiset of elements (i.e., nothing is added or dropped)
  - Sets and strings: |AB| = |A| + |B|
  - Cart totals: never negative
  - Cart: membership tier discount never increases price!

```
fc.assert(fc.property(fc.array(Item), fc.tuple(Tier,Tier), (items,[tLow,tHigh]) => {
  const order = ["guest","bronze","silver","gold"];
  const low = computeTotal(items, tLow);
  const high = computeTotal(items, tHigh);
  return order.indexOf(tHigh) >= order.indexOf(tLow) ? high <= low : true;
}));
```

Common Property: Metamorphic

- Property holds under **input transformations**
  - Sorting: reversing/shuffling input does not change the sorted output
  - Cart: reordering items does not change the total
  - JSON parsing: reordering object keys produces the same parsed result
  - Shipping: changing address / ZIP doesn't affect the price

```
fc.assert(fc.property(fc.array(fc.integer()), xs => {
  const a = doubleBogosort(xs);
  const b = doubleBogosort(xs.reverse());
  return a.length === b.length && a.every((v,i)=> v === b[i]);
}));
```

Common Property: Differential

- Two implementations (or versions) **agree on outputs and errors**
  - Slow but trusted reference solution vs. optimized version
  - Old vs. new implementation after refactoring
  - Third-party library vs. your code

```
fc.assert(fc.property(fc.array(fc.integer()), xs => {
  const a = doubleBogosort(xs);
  const b = radixSort(xs);
  return a.length === b.length && a.every((v,i)=> v === b[i]);
}));
```

In-class exercise part 2: Properties in NodeBB, Wordle, or RoboPiper

Conditional Properties

Conditional Properties

- **If** condition A is met, **then** property B must hold.
  Mathematical logic: $A \Rightarrow B$
- Examples:
  - **Wordle**: If the guess word matches the secret word the output should be (G,G,G,G,G).
  - **Compilers**: If the program includes an invalid keyword, the parser should throw a parse error.
  - **NodeBB** converts post titles into URL-safe "slugs" (e.g., "Hello World!" becomes hello-world). If the input string contains at least one alphanumeric character then the generated slug must never be empty and must only contain a-z, 0-9, and -.

Conditional Properties Approach 1: Filtering

Most PBT frameworks allow you to "discard" inputs that don't meet your criteria. In fast-check you use the fc.pre() method.

- **How it works:** The generator creates a random withdrawal amount. If that amount is ≤ the balance, the test framework simply throws that case away and tries again with a new random number.
- **The Pro:** It's very easy to write.
- **The Con:** It can be inefficient. If your condition is very rare (e.g., "only when the amount is exactly 1,000,000.03"), the framework might exhaust its retry limit before finding enough valid cases.

Conditional Properties Approach 2: Targeted Generators

Instead of generating *any* number and throwing away the bad ones, you create a custom generator that *only* produces "invalid" inputs.

- **How it works:** You define a generator specifically for the error case: amount = random_int(min=balance + 1).
- **The Pro:** Every single test run is a "hit." It’s incredibly efficient and ensures you are testing the error handling thoroughly.
- **The Con:** It requires more setup code and a deeper understanding of your data constraints.

Conditional Properties Approach 3: Branching Properties

- **How it works:** The test itself has branches (e.g. if and else) and the property is defined across the branches.
- **The Pro:** This creates a "complete" specification of the function in one place.
- **The Con:** It can make the test code more complex and harder to debug if it fails.

Up next ...

- Practice using PBT with fast-check this Monday in Recitation
- More PBT including ”vibe analysis” and more on input-generators on Tuesday.