Testing:
From Examples to Properties

17-313: Foundations of Software Engineering

https://cmu-313.github.io

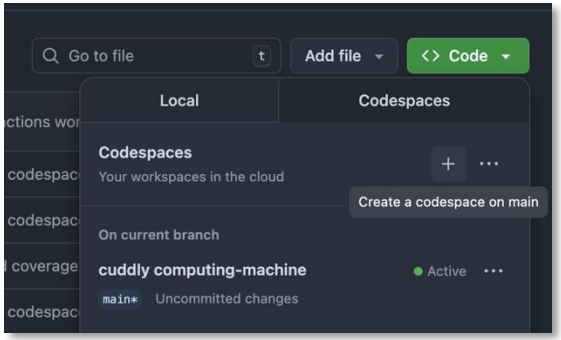
Michael Hilton and **Chris Timperley**

Fall 2025

Administrivia

- No class next Tuesday (Democracy Day)

Recap: Setup



- Everyone should participate on their laptop
- Open **CMU-313/PierogI/O** in **Codespaces**
  - https://github.com/CMU-313/PierogIO
- Create a **branch** for this activity
  - `git checkout -b andrew-id/tests`
  - `git push -u origin andrew-id/tests`
- Add your branch name to the spreadsheet
  - http://bit.ly/3WqXBBe

Recap: Write tests to find bugs

- Find bugs in the implementation by writing test cases
  - “npm run test” to run the tests (or hit the run test button in the IDE)
- Fix any bugs that you find!
  - every bug should have a corresponding regression test
  - **only start fixing the bug once you have written the test**
- Push the changes to your branch to GitHub
  - `git push -u origin andrew-id/tests`
- When you have written at least one test, fixed a bug, and pushed your changes to GitHub, **update the spreadsheet**

 Activity: Let's break things

- Now, you will inject a silent bug into someone else's code that **IS NOT caught** by the test suite **(i.e., the tests don't fail)**
- Use the spreadsheet to takeover someone else's branch
  - Mark as **"In Progress"** when you claim it
  - Mark as **"Done"** when you have added a silent bug
- Switch to that person's branch
  - `git pull`
  - `git checkout {the-branch-name}`
- **If you finish early, work on a second, unclaimed branch!**

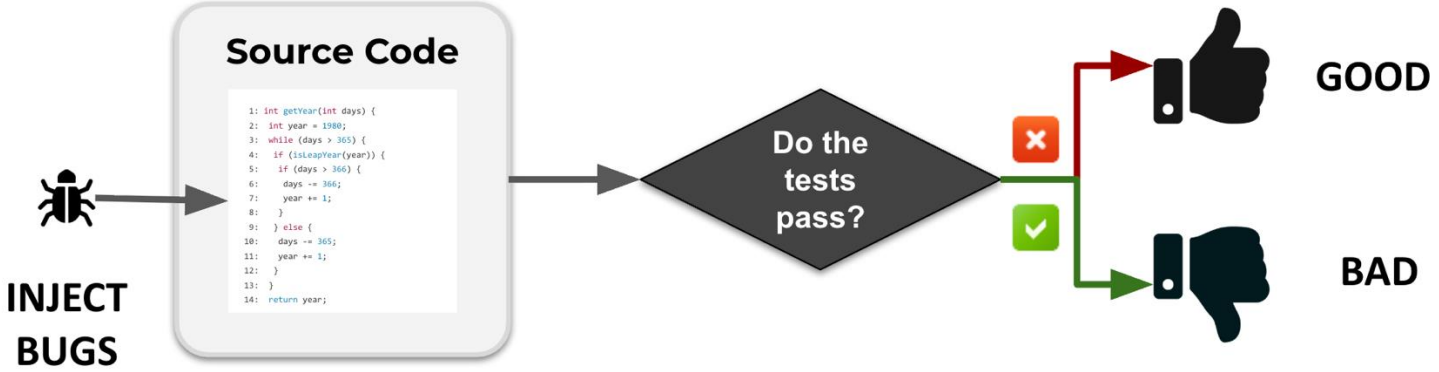
Activity: Bolster your tests + asserts

- Go **to any branch marked as "Done"** and pull the breaking changes
  - `git checkout {your-branch}`
  - `git pull`
- **Without fixing the code, can you add an assertion to the program or the tests that catches the bug?**

How can we measure the strength of our tests?

Mutation Testing measures test adequacy

- Faults (known as **mutations**) are automatically planted into your code, and your tests are then run
  - mutation score measures how many mutants are “killed” by your tests
  - good test suites have higher mutation scores



Mutations mimic common mistakes

- Conditionals Boundary
- Increments
- Invert Negatives
- Math
- Negate Conditionals
- Return Values
- Void Method Calls
- Empty returns
- False Returns
- True returns
- Null returns
- Primitive returns
- Remove Conditionals
- Experimental Switch
- Inline Constant
- Constructor Calls
- Non Void Method Calls
- Remove Increments
- Experimental Argument Propagation
- Experimental Big Integer
- Experimental Member Variable
- Experimental Naked Receiver
- Negation
- Arithmetic Operator Replacement
- Arithmetic Operator Deletion
- Constant Replacement
- Constant Replacement
- Bitwise Operator
- Relational Operator Replacement
- Unary Operator Insertion

Mutations mimic common mistakes

```
 1: int getYear(int days) {
 2:  int year = 1980;
 3:  while (days > 365) {
 4:   if (isLeapYear(year)) {
 5:    if (days > 366) {
 6:     days -= 366;
 7:     year += 1;
 8:    }
 9:   } else {
10:    days -= 365;
11:    year += 1;
12:   }
13:  }
14:  return year;
15: }
```

```
if (days > 366)
if (days >= 366)
```

```
int year = 1980;
int year = 1981;
```

```
int year = 1980;
int year = -1981;
```

```
if (days > 366) {
  days -= 366;
  year += 1;
}
```

```
if (days > 366)
if (!(days > 366))
```

```
return year;
return 0;
```

```
year += 1;
```

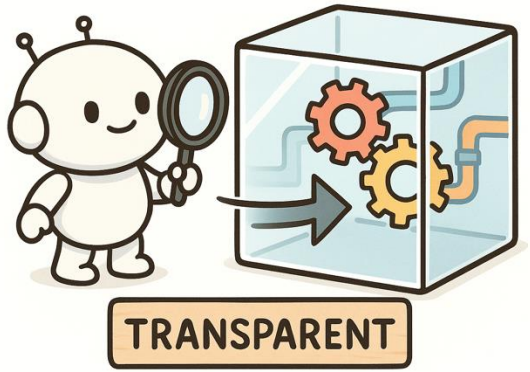
```
year += 1;
year -= 1;
```

Pros and Cons: Mutation Testing

- It measures the ability of your test suite to find bugs!
  - warning: it's only as good as your set of mutations
  - small or unrepresentative mutant sets don't tell you much
- It doesn't tell you about **errors of omission**
  - E.g., missing validation
- It can be **extremely expensive** for large systems
  - we may need to execute the whole test suite for each mutant
  - useful for small, cheap, and fast unit tests
  - run it on your PRs — not the whole codebase

Transparent Testing (Whitebox)

- **Observation: Many of you first read the code, then designed an input to hit the buggy line.**
- **Why this can be tricky …**
  - You may **not have source** (libraries/services)
  - **Bias**: knowing the code nudges you to test what you can see, not what users do.
  - **Commission vs. omission**: easy to check the **wrong** thing; hard to notice **missing** checks.
  - **Reachability**: crafting inputs to deep / rare paths is **hard** (state, constraints).

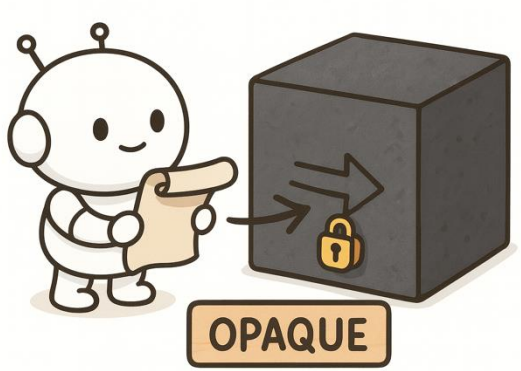



```
* @param {Object} order - The order object with items array
* @param {Object} context - Context containing profile, delivery, and optional coupon
* @returns {number} - Total cost in cents
*/
function total(order, context) {
  const { profile, delivery, coupon = null } = context;

  const orderSubtotal = subtotal(order);
  const orderDiscounts = discounts(order, profile, coupon);
  const orderDelivery = deliveryFee(order, delivery, profile);
  const orderTax = tax(order, delivery);
  let orderTotal = orderSubtotal - orderDiscounts + orderDelivery + orderTax;

  if (delivery.rush) {
    orderTotal += 299;
  }

  if (orderTotal > 10000) {
    const formatted = (orderTotal / 100).toFixed(2);
    orderTotal = formatted + "00";
    orderTotal = parseInt(orderTotal);
  }
```

image credit: GPT-5

Opaque Testing (Blackbox)

- **Write tests without peeking at the code.** Design based on inputs and expected outputs (e.g., from APIs, documentation)



- **Techniques:**
  - partition inputs into **equivalence classes** and pick representatives from each one
  - explore **boundary values** (e.g., minimum, just-below, just-above, max)
  - try **error cases** and **negative inputs** (e.g., empty, null, NaN, duplicates)
  - …

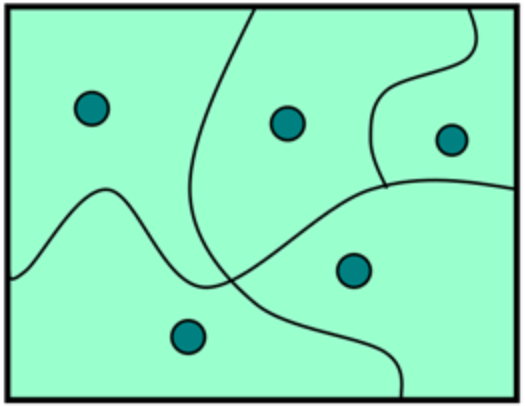


image credit: GPT-5

🎯 Let's Test Smaller Targets

- The **`total`** method is complex and has lots of dependencies!
  - **`deliveryFee`**, **`subtotal`**, **`tax`**, **`discounts`**
- If total fails, there could be a bug in any of these dependencies or in the code that ties them together (i.e., the code in **total**)
  - this makes it hard to debug failures (we have more lines to go through)
  - good tests should fail for a single reason
- Instead, we can test those dependencies directly
  - fewer reasons for failure; fewer lines to debug / inspect
  - easier to write tests

Translucent Testing (Greybox)

- If we have access to the code, we can **measure properties of the code** without looking directly at the code itself
  - i.e., coverage, mutation score
- We can use those measures to **assess test adequacy** and help to **find weaknesses** in our test designs
- **In practice, we often do a combination of all types of testing**

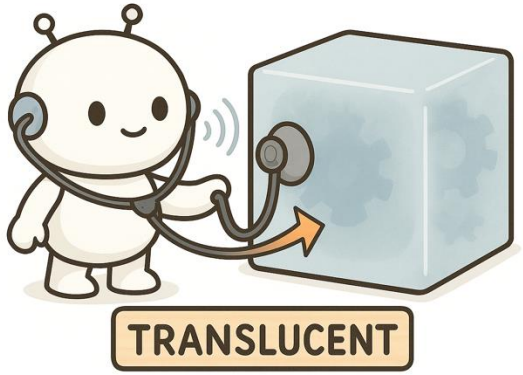


image credit: GPT-5

% Coverage report from v8

File	% Stmts	% Branch	% Funcs	% Lines
All files	98.22	91.48	100	98.22
delivery.js	97.33	88.88	100	97.33
discounts.js	96.15	88.88	100	96.15
subtotal.js	100	100	100	100
tax.js	100	100	100	100
total.js	100	100	100	100

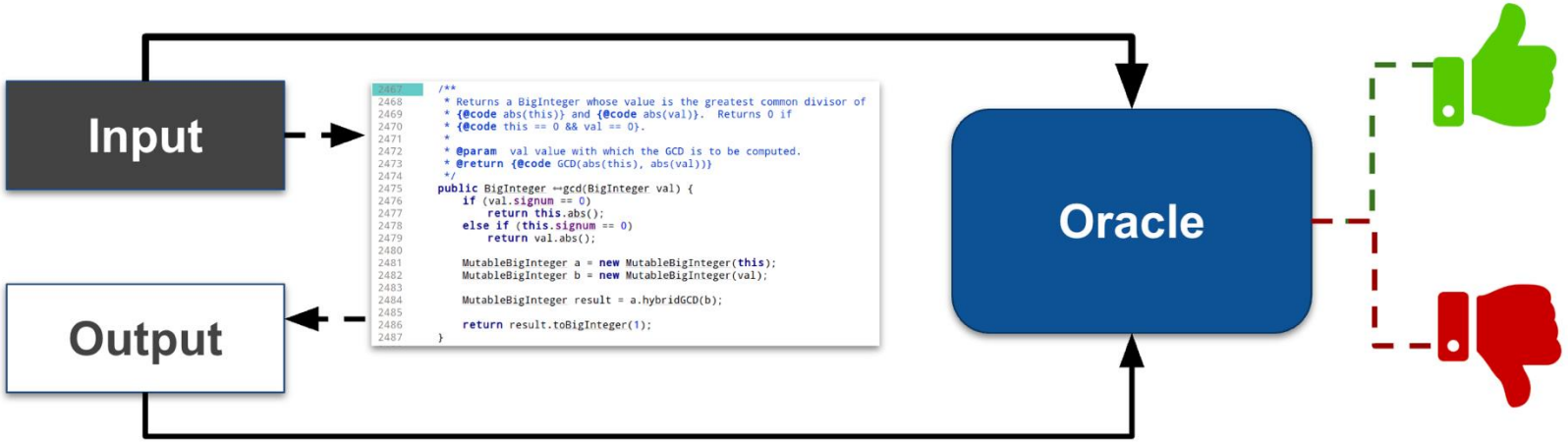
✅ Let's test different order contexts

- Let's test `total` using the **same order** in **different contexts**
  - `Tier: “guest” | ”regular” | “vip”`
  - `Zone: “local” | “outer”`
  - `Rush: true | false`
  - `Coupon: null |”PIEROGI-BOGO” | ”FIRST10”`
- Our order
  - One 6-pack of Potato pierogies (P6-POTATO)
- We can use a **table-driven test** (`.forEach`) and a **helper function** (`createContext`) to create them with minimal code!

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



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

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 nonDecreasing(ys);
}));
```

We want to test our implementation of an amazing sorting algorithm

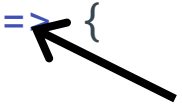
Property-Based Testing replaces single examples with properties that hold for many inputs

```
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 nonDecreasing(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 nonDecreasing(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 nonDecreasing(ys);
}));
```

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

Common Properties: Preservation

- 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 Properties: 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 Properties: 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]);
}));
```

Activity: Devise a set of properties

- Switch to the **“properties” tab** of the spreadsheet
  - http://bit.ly/3WqXBBe
- Create **a new branch** (from main) for property-based testing and add your branch name to the spreadsheet
  - `git checkout main`
  - `git checkout -b andrew-id/properties`
  - `git push -u origin andrew-id/properties`

Activity: Devise a set of properties

- Try to come up with a set of properties that you could test for the codebase (e.g., Preservation, Metamorphic, Differential)
  - discuss with your neighbors!
- Add your property ideas to the implementation code
  - either as part of the docstring or as a comment
- **Push your changes to GitHub and update the spreadsheet**

```
/**
* 1. SUBTOTAL: Base item prices plus add-ons
*    - Add-ons: sour cream ($0.99), fried onion ($1.49), bacon bits ($1.99)
*    - Add-on prices are per pack (multiplied by quantity)
* 2. DISCOUNTS:
*    - Volume discounts: Applied automatically based on quantity per item
*      • 12-pack: 5% off
*      • 24-pack: 10% off
*    - Coupon codes (see below)
* 3. DELIVERY:
*    - Local zone (0-10 km): $3.99
*    - Outer zone (10+ km): $6.99
*    - Rush delivery: +$2.99 surcharge
*    - Free delivery thresholds (based on discounted subtotal):
*      • Guest: $50 or more
*      • Regular: $40 or more
*      • VIP: $30 or more
*    - When free delivery applies, only rush fee is charged (if applicable)
* 4. TAX:
*    - 8% sales tax on hot items only
*    - Frozen items are tax-exempt (0% tax)
*    - Delivery fee is taxable only if the order contains any hot items
*/
```

Key Takeaways

- Test adequacy is measurable (coverage, mutation score)
- Mutation testing is great for **cheap, fast unit tests**
- White/black/grey-box are complementary, not competing
- Property-based tests exercise **many inputs** with **one property**
- Strong oracles are the bottleneck: use **Preservation**, **Metamorphic**, **Differential** patterns