Software Testing

Running example: **TerpLedger**, a shared-expense ledger · plus quicksort, BSTs, and triangles

Learning Objectives

1. **Explain** what testing can and cannot show: verification vs. validation, testing vs. debugging and formal verification
2. **Place** unit, integration, system, acceptance, and regression tests on the V model and test pyramid
3. **Design** test cases with equivalence partitioning, boundary values, and an explicit **oracle**
4. **Write** `pytest` tests: assertions, parametrization, fixtures, test doubles; apply **TDD**
5. **Write** property-based tests with **Hypothesis**; explain generators, shrinking, differential testing
6. **Measure** statement / branch / condition coverage, and explain why 100% ≠ correct
7. **Use** mutation testing and fuzzing to grade and stress a test suite
8. **Run** tests in CI, and **use LLMs** to write tests *without trusting them blindly*

Two Warnings from the Pioneers

> *"Program testing can be used to show the **presence** of bugs, but never to show their **absence**!"* (Edsger Dijkstra)

A function of two 64-bit ints has 2^{128} inputs. You will test about 20 of them. Everything today is a way to **choose** those 20 better, **check many more** automatically, or **measure** how much your tests check.

> *"There are two ways of constructing a software design: make it so simple that there are **obviously no** deficiencies, or so complicated that there are **no obvious** deficiencies. The first method is far more difficult."* (Tony Hoare)

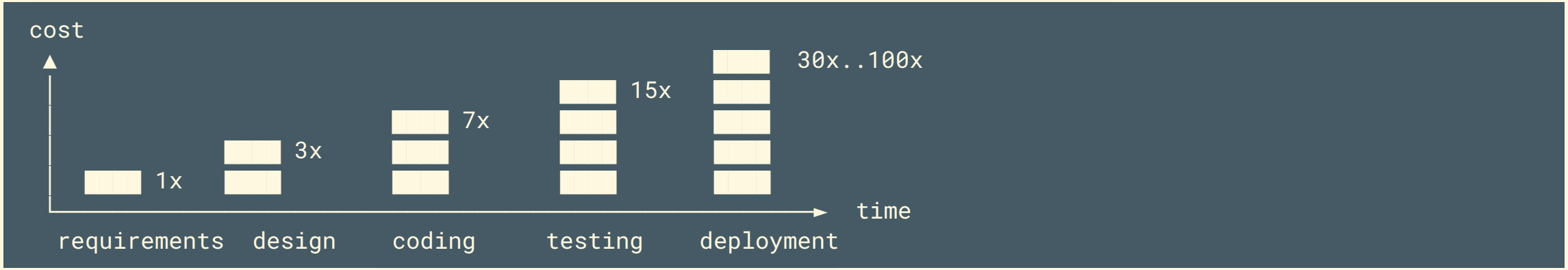
Code that is hard to test (hidden globals, network, clock) usually has a design problem.

Why Testing Matters

- **Bugs are expensive.** Developers spend an estimated **~50%** of their time finding and fixing bugs. Testing is the **main** way industry evaluates software.
- **Software is built by changing it.** A *regression* is an old feature broken by a new change. A test suite is a regression alarm.
- **Tests are executable documentation.** Unlike a comment, a test fails when the behavior it describes changes.
- **GenAI makes testing *more* important.** An LLM writes 100 plausible lines in seconds. Before you ship them, ask:

> ***How would I know if this is wrong?*** Every technique today answers that question.

The Cost of Finding Bugs Late



A unit test on your laptop finds the bug in **a minute**. When a customer finds it, you need a bug report, a reproduction, a fix, a review, a release, and you may lose data or trust.

> **Shift left:** test early, test automatically, test on every change.

Vocabulary

> A **test case** has an **input** (plus setup), an **action** that runs the code, and an **expected result**. The **test oracle** is whatever decides the expected result: a person, a spec, or another program.

Verification: *are we building the product right?*
Does `split_evenly` return shares that sum to the total, as the docstring says?

Validation: *are we building the right product?*
Do club members *want* the leftover cent to go to the first person?

Testing finds ***that*** something is wrong.
Debugging finds ***why*** and fixes it.

A small, focused failing test points at the bug. Once the bug is fixed, the test stays in the suite as a **regression test**.

Why Not Just Prove It? Formal Verification in Dafny

```
// Weak spec
method Max2(a: int, b: int) returns (r: int)
  ensures r >= a && r >= b
{
  if a > b { r := a; }
  else     { r := b + 1; }   // BUG
}
```

```
$ dafny verify max.dfy
... 3 verified, 0 errors     ✅ ?!
```

The buggy `Max2` **verifies**: `b + 1` really is ≥ both inputs.

The spec is missing `ensures r == a || r == b`. With that line added, Dafny rejects the bug.

> A proof is only as good as its **specification**, just as a test is only as good as its **oracle**.

What Does "Sorted" Mean?

```
method SortWeak(s: seq<int>) returns (r: seq<int>)
  ensures Sorted(r)
{ r := []; }            // sorted! ...and useless

method Sort(s: seq<int>) returns (r: seq<int>)
  ensures Sorted(r)                    // in order
  ensures multiset(r) == multiset(s)   // permutation
```

- "Sorted" alone accepts `[]`, and accepts `[1,4,3,4,5] → [1,3,3,4,5]`
- A correct spec needs **two** properties: ordered **and** a permutation
- The verified insertion sort needs **5 loop invariants + 3 assert hints** for a 10-line algorithm

> Verification gives correctness for **every** input, but you pay twice: once to write the spec, again to write the proof. **Testing** is the practical middle ground. We'll meet "sorted + permutation" again with Hypothesis. *(More: CMSC433)*

Running Example: TerpLedger

A shared-expense ledger for a student club.

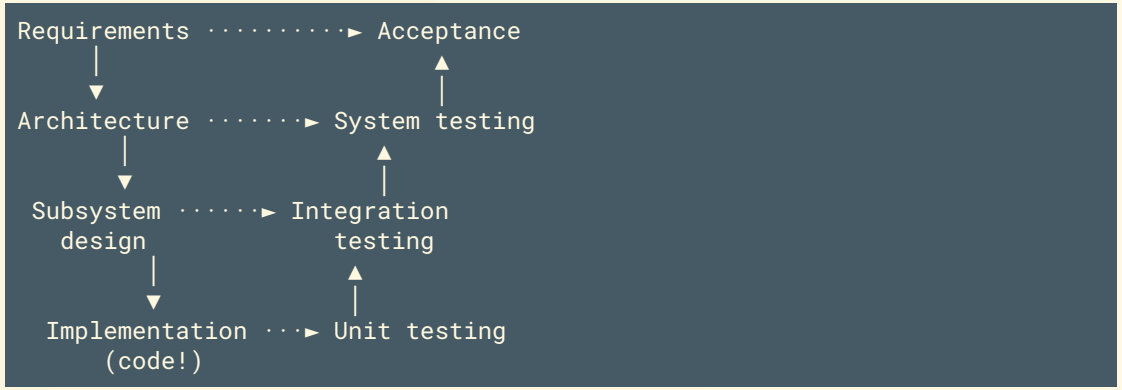
`2026-09-12,Pizza night,2350,alice,alice;bob;carol`

Module	What it does	Main technique
`app/split.py`	split money in **integer cents**	unit, property, differential, mutation
`app/parser.py`	parse the one-line format	test design, fuzzing, round-trip
`app/ledger.py`	in-memory + SQLite stores	contract (differential) tests
`app/categorize.py`	asks an **LLM** for a category	test doubles
`app/main.py`	FastAPI web API	integration tests

```python
def split_evenly(total_cents: int, n_people: int) -> list[int]:
    """Shares sum to total; differ by ≤ 1 cent; first people absorb leftovers."""
    if n_people < 1:    raise ValueError("need at least one person")
    if total_cents < 0: raise ValueError("total must be non-negative")
    base, leftover = divmod(total_cents, n_people)
    return [base + 1 if i < leftover else base for i in range(n_people)]
```

Levels of Testing and the V Model

- **Unit:** one function or class.
  `split_evenly(1000,3) == [334,333,333]`
- **Integration:** components together.
  POST a transaction → `/balances` updates
- **System (E2E):** the whole app, from the user's view, in a browser
- **Acceptance:** does it meet user needs? The treasurer checks a month against the bank statement (**validation**)



> Every decision you make while building has a test level that checks it.

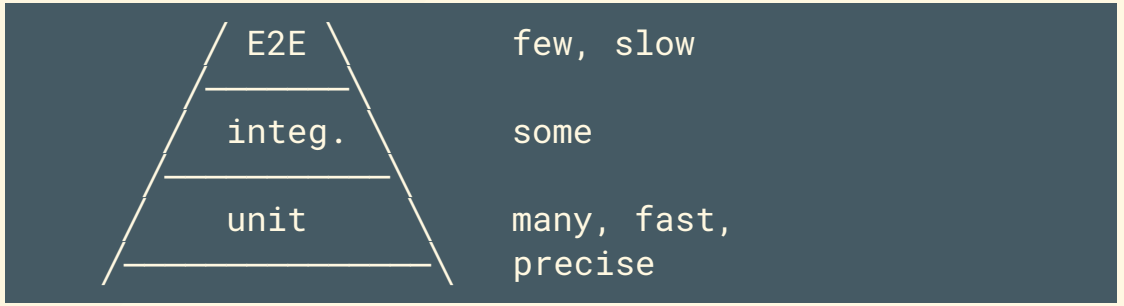
Regression Testing and the Test Pyramid

A teammate (or an LLM) "simplifies" `split_evenly`:

```
return [total_cents // n_people] * n_people
```

```
FAILED test_split_gives_leftover_cents_to_first_people
FAILED test_split_examples[1-2-expected0]
FAILED test_split_examples[5-4-expected1]
FAILED test_split_examples[200-6-expected3]
4 failed, 9 passed in 0.02s
```

Without the suite, the "simplification" would make money quietly disappear.



Unit tests are cheap and **point at the broken function**. E2E tests are slow, flaky, and vague about where the failure is.

Testing by Purpose, and When to Write Tests

Type	Question	TerpLedger example
Functional	right result?	balances sum to zero
Performance	fast enough?	`/balances` < 100 ms with 100k transactions
Security	can it be misused?	a malformed import line can't crash the server
Usability	can users use it?	a new member records a purchase unaided
Availability	is it up?	`/health` answers after a deploy

- **Test first (TDD):** write a failing test, then the code. *(later today)*
- **Test after:** common, with a trap: it tends to test what the code **does**, not what it **should** do
- **Iterate:** retest and refactor, with the suite as your safety net

Designing Tests: A Warm-up

```
def withdraw(balance, amount):
    if amount <= balance:
        return balance - amount
    return balance
```

Balance	Amount	Expected	Why?
100	50	50	typical
100	100	0	boundary: withdraw everything
100	101	100	boundary: one cent too much
100	0	100	boundary: zero
100	-10	**?**	invalid input → code returns **110**

> Is a negative withdrawal turning into a deposit a bug? Nobody decided. **Testing exposes missing requirements, not just bugs.**

▲ Exercise: The Triangle Problem

A program reads **three side lengths** and reports **equilateral**, **isosceles**, **scalene**, or **not a triangle**.

Write down 10 test cases. (3 minutes, with a neighbor.)

Triangle: Partition the Input Space

Valid triangles: one per type

a	b	c	Expected
5	5	5	equilateral
5	5	6	isosceles
3	4	6	scalene

Invalid: largest side in *each* position

a	b	c	Expected
2	3	6	not a triangle
10	2	3	not a triangle
1	10	2	not a triangle

Boundaries: where sums are *equal*

a	b	c	Expected
2	2	4	not a triangle
5	5	9.999	isosceles
5	5	10	not a triangle

Bad input

a	b	c	Expected
0	2	2	invalid input
-1	2	2	invalid input
10^6	10^6	10^6	equilateral

Triangle: What Most People Miss

```
def is_triangle(a, b, c):
    if a <= 0 or b <= 0 or c <= 0:
        return False
    return a + b > c and a + c > b and b + c > a   # ALL three, strictly
```

- **Only one inequality checked?** `(2, 3, 6)` passes and `(10, 2, 3)` doesn't. Permute the longest side.
- `>=` **instead of** `>`**?** `(2, 2, 4)` is a flat line. Test where the sums are **equal**.
- **Floats:** `is_triangle(0.1, 0.2, 0.3)` returns **True**, because `0.1 + 0.2 == 0.30000000000000004`

> These are the two systematic techniques: **equivalence partitioning** (which classes of input?) and **boundary value analysis** (where are the edges?).

Equivalence Partitioning and Boundary Values

Partitioning: split the input domain into classes the program should treat the same way. Test at least one input per class. **Boundary analysis:** test at, just below, and just above each edge. Bugs live in `<` vs `<=`.

Parser amount field (positive integer cents):

Partition	Input	Expected
positive int	`2350`	accepted
zero	`0`	`ParseError`
negative	`-5`	`ParseError`
decimal	`12.50`	`ParseError`
not a number	`abc`	`ParseError`
empty	`""`	`ParseError`

`split_evenly` **boundaries:**

- `n_people`: **0** (error), **1** (gets everything), 2
- `total_cents`: **-1** (error), **0** (all zeros), **1**
- leftover: none (divisible), and the maximum `n - 1`

Test **both sides**: 0 rejected **and** 1 accepted. (Mutation testing will catch TerpLedger forgetting this.)

Positive, Negative, and the Test Oracle

- **Positive tests:** valid input gives the right result. **Negative tests:** invalid input raises the *documented* error. Beginners write mostly positive tests; bugs and security holes are mostly in the negative cases.
- **Edge cases:** empty lists, a single element, huge numbers, duplicates, Unicode names, leap days

Where does the expected value come from?

Oracle	Example	Technique
the **specification**	compute `[334, 333, 333]` by hand	unit tests
a **reference implementation**	slow, obviously-correct version	differential
a **property**	balances sum to zero	property-based
implicit	"it doesn't crash"	fuzzing

> **Never** copy the code's output into the test. Then the code is its own oracle, and the test passes forever.

pytest: Assertions and Arrange–Act–Assert

```python
from app.split import split_evenly

def test_split_divides_evenly_when_possible():
    assert split_evenly(900, 3) == [300, 300, 300]

def test_split_gives_leftover_cents_to_first_people():
    assert split_evenly(1000, 3) == [334, 333, 333]

def test_split_single_person_gets_everything():
    assert split_evenly(2350, 1) == [2350]

def test_split_zero_total_is_all_zeros():
    assert split_evenly(0, 4) == [0, 0, 0, 0]
```

```
    def test_split_gives_leftover_cents_...():
>       assert split_evenly(1000, 3) == [334, 333, 333]
E       assert [333, 333, 333] == [334, 333, 333]
E         At index 0 diff: 333 != 334
tests/test_split.py:21: AssertionError
```

- Files `test_*.py`, functions `test_*`, plain `assert`
- **Arrange** inputs → **Act** (call) → **Assert**
- Name tests after **behaviors**, not `test_split_1`

Parametrized Tests: a Partition Table as Code

```python
@pytest.mark.parametrize("bad_line", [
    "",                                     # no fields
    "2026-09-12,Pizza,2350,alice",          # only 4 fields
    "yesterday,Pizza,2350,alice,bob",       # bad date
    "2026-02-30,Pizza,2350,alice,bob",      # impossible date
    "2026-09-12,Pizza,12.50,alice,bob",     # not integer cents
    "2026-09-12,Pizza,0,alice,bob",         # zero amount
    "2026-09-12,Pizza,2350,alice,bob;;carol",   # empty name in list
])
def test_parse_rejects_malformed_input(bad_line):
    with pytest.raises(ParseError):
        parse_transaction(bad_line)
```

> Name the **specific** exception. `pytest.raises(Exception)` also passes on an unrelated `IndexError` crash. This weak assertion shows up a lot in AI-generated tests.

Fixtures: Setup and Teardown

```python
@pytest.fixture
def client():
    # fresh app per test → no shared state
    return TestClient(create_app(llm=KeywordLLM()))

def test_health(client):   # injected by name
    assert client.get("/health").json() == {"status": "ok"}

@pytest.fixture
def db():
    ledger = SqliteLedger(":memory:")  # setup
    yield ledger                       # test runs here
    ledger.close()                     # teardown, always
```

- A test **requests** a fixture by naming it as a parameter
- Code after `yield` runs even if the test **failed**
- Built-ins: `tmp_path`, `monkeypatch`
- Run subsets: `pytest -k leftover`, `pytest -m live`

> If tests share one app, results depend on **test order**. That's how you get flaky tests.

Test Doubles: Replacing What You Can't Use in a Test

Some dependencies are slow, cost money, or aren't deterministic: payment APIs, the clock, a database server, **an LLM**.

Double	What it does	TerpLedger
stub	canned answers, no logic	LLM that always says `"food"`
fake	simplified working implementation	in-memory `Ledger`; `KeywordLLM`
spy	stub/fake that **records** its calls	`ScriptedLLM` records every prompt
mock	expectations; fails if violated	LLM that raises if ever called

A stub tests what your code does **with an answer**. A spy tests what it **asked**. A mock tests what it must **never** do. A fake lets a whole subsystem run.

A Spy LLM, Injected Through a Seam

```python
class ScriptedLLM:
    """Pre-scripted replies; records every prompt."""
    def __init__(self, *replies):
        self._replies = list(replies)
        self.prompts = []

    def complete(self, prompt):
        self.prompts.append(prompt)
        if not self._replies:
            raise AssertionError("unexpected extra call")
        return self._replies.pop(0)
```

- Tests the messy replies (`"Food."`, `"groceries"`) **deterministically**, in milliseconds, with no model running
- **Seam = dependency injection:** `create_app(ledger=..., llm=...)` exists so tests can pass in fakes

> "More mocking = better tests" is **wrong**. Mock-everything tests check call order, not behavior. Prefer real objects, then fakes, and mock only boundaries you don't own.

What Makes a Test Good?

- **Fast:** milliseconds. TerpLedger's 74 tests run in < 1 s
- **Isolated:** no dependence on other tests, the network, or shared state
- **Deterministic:** same code, same verdict. **Flaky** tests teach people to ignore red
- **One behavior per test:** a failure names the broken fact
- **Tests behavior, not implementation:** check what it returns, not how
- **Seen to fail:** break the code once and watch the test go red

> A test you have never seen fail is a test you don't know works.

Test-Driven Development: Red → Green → Refactor

```python
# RED: write the tests first
def test_total_paid_by_sums_only_that_payer():
    txs = [_tx(2350, "alice"), _tx(500, "bob"),
           _tx(150, "alice")]
    assert total_paid_by(txs, "alice") == 2500

def test_total_paid_by_unknown_person_is_zero():
    assert total_paid_by([_tx(2350, "alice")], "zoe") == 0
```

```
E   ImportError: cannot import name 'total_paid_by'
```

```python
# GREEN: just enough code
def total_paid_by(transactions, person):
    return sum(t.amount_cents for t in transactions
               if t.payer == person)
```

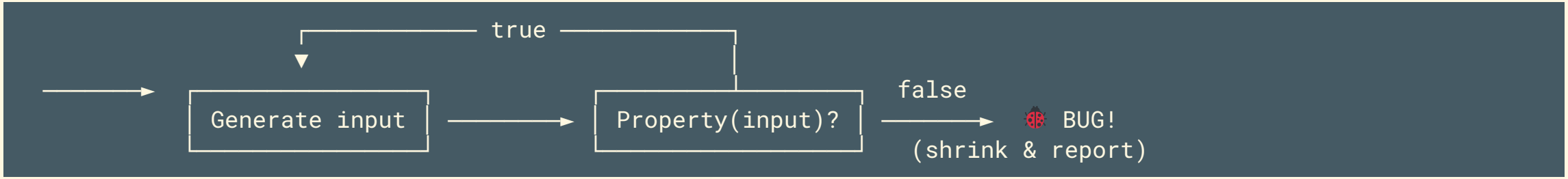
```
2 passed in 0.05s
```

- Writing the 2nd test **forced a design decision**: someone who never paid owes 0; it's not an error
- Every test has been **seen to fail**
- **Refactor** with the tests staying green

> TDD is **not** "write all the tests up front." It's one small test at a time, each followed right away by the code that passes it.

Property-Based Testing

> A **property** is a boolean condition that must hold for **every** input in a domain. A **generator** produces inputs. The framework generates many random inputs, checks the property on each, and reports any input that makes it false.



A unit test is a **point check**; a property is a **law**. You stop picking inputs (you're biased) and state what must always be true. Origin: **QuickCheck** (Haskell, 2000). Now available as QCheck, jqwik, fast-check, and **Hypothesis** (Python).

A First Property: Reverse

```
from hypothesis import given, strategies as st

@given(xs=st.lists(st.integers()))
def test_reverse_twice_is_identity(xs):
    assert reverse(reverse(xs)) == xs
```

Now a **buggy** reverse:

```
def reverse_buggy(xs):
    return list(xs)       # doesn't reverse at all!
```

Reverse-twice **still passes** all 100 runs, because doing nothing twice is still doing nothing.

A plain unit test catches it at once:

```
assert reverse_buggy([1, 2, 3]) == [3, 2, 1]
```

A **stronger** property:

```
@given(l1=st.lists(st.integers()), x=st.integers(),
       l2=st.lists(st.integers()))
def test_reverse_distributes(l1, x, l2):
    assert reverse(l1 + [x] + l2) == \
           reverse(l2) + [x] + reverse(l1)
```

```
Failing test case: l1=[], x=0, l2=[1]
```

> A property is only as good as what it says. Properties **complement** unit tests; they don't replace them.

Example: Quicksort (Claude-written) DEMO

```python
def quicksort(items):
    if len(items) <= 1:
        return items[:]
    pivot = items[len(items) // 2]
    less    = [x for x in items if x < pivot]
    equal   = [x for x in items if x == pivot]
    greater = [x for x in items if x > pivot]
    return quicksort(less) + equal + quicksort(greater)

@given(st.lists(st.integers()))
def test_quicksort_is_sorted(items):
    assert is_sorted(quicksort(items))

@given(st.lists(st.integers()))
def test_quicksort_is_permutation(items):
    assert Counter(quicksort(items)) == Counter(items)
```

- Claude's **10 unit tests** (empty, single, duplicates, negatives…) all pass. **Is that enough?**
- Swap `less` / `greater` → `is_sorted` finds a counterexample at once
- Is "sorted" the whole spec? **No.** `return []` is sorted. We also need **permutation**, the same lesson as the Dafny slide

Hypothesis Hunts a TerpLedger Bug

```
totals      = st.integers(min_value=0, max_value=10_000_000)
group_sizes = st.integers(min_value=1, max_value=200)

@given(total=totals, n=group_sizes)
def test_shares_sum_to_total(total, n):
    """Conservation of money: not a cent appears or disappears."""
    assert sum(split_evenly(total, n)) == total
```

Against the "simplified" `[total // n] * n`:

```
E       assert 0 == 1
E        +  where [0, 0] = split_evenly(1, 2)
E       Failing test case: test_shares_sum_to_total(total=1, n=2)
```

> Nobody picked the inputs, and the counterexample is the **simplest** one: split 1 cent between 2 people. The property never says what `split_evenly(7, 3)` *equals*. Properties **constrain** the output without fixing its value.

Shrinking

> **Shrinking** automatically simplifies a failing input to the smallest input that **still fails**.

Property: *"the total of a list of amounts is never negative"*

```
  [123, -999999999999, 5, 0, 5]  fails
→ [123, -1, 5, 0, 5]             fails
→ [123, -1]                      fails
→ [-1]                           fails
→ []                             passes ✗
Report: amounts=[-1]
```

Strategies depend on the type:

- **Numbers:** smaller magnitude, or 0
- **Lists:** remove or shorten elements
- **Strings:** shorter, simpler characters
- **Custom objects:** shrink fields recursively. For **trees**: shrink node values, remove subtrees, replace a node by a subtree

Debugging "8,392,107 cents among 173 people is 134 short" is hard. Debugging `(1, 2)` is easy.

Generators (Strategies) Compose

```python
names = st.text(alphabet="abcdefghijklmnopqrstuvwxyz", min_size=1, max_size=8)

transactions = st.builds(
    Transaction,
    date=st.dates(min_value=date(2000, 1, 1), max_value=date(2099, 12, 31)),
    description=names,
    amount_cents=st.integers(min_value=1, max_value=1_000_000),
    payer=names,
    participants=st.lists(names, min_size=1, max_size=10).map(tuple),
)

@given(txs=st.lists(transactions, max_size=20))
def test_balances_sum_to_zero(txs):
    assert sum(balances(txs).values()) == 0
```

- Good generators are **not uniform**: Hypothesis over-samples 0, 1, min/max, `[]`, and odd Unicode
- `@st.composite` builds custom strategies, e.g. a BST made by repeated `insert` *(next slide)*

How to Specify It: Properties of a BST DEMO

```python
@given(bst())                                    # TEST YOUR GENERATOR first!
def test_arbitrary_valid(t):          assert valid(t)

@given(KEYS, VALUES, bst())                      # VALIDITY: ops return valid trees
def test_insert_valid(k, v, t):       assert valid(insert(k, v, t))

@given(KEYS, VALUES, bst(), KEYS)                # POSTCONDITION: relate result to args
def test_insert_post(k, v, t, k2):
    assert find(k2, insert(k, v, t)) == (v if k == k2 else find(k2, t))

@given(KEYS, VALUES, KEYS, VALUES, bst())        # METAMORPHIC: related calls, related results
def test_insert_insert(k1, v1, k2, v2, t):
    assume(k1 != k2)
    assert equiv(insert(k1, v1, insert(k2, v2, t)), insert(k2, v2, insert(k1, v1, t)))

@given(KEYS, bst())                              # MODEL-BASED: compare against a sorted list
def test_delete_model(k, t):
    assert to_list(delete(k, t)) == model_delete_key(k, to_list(t))
```

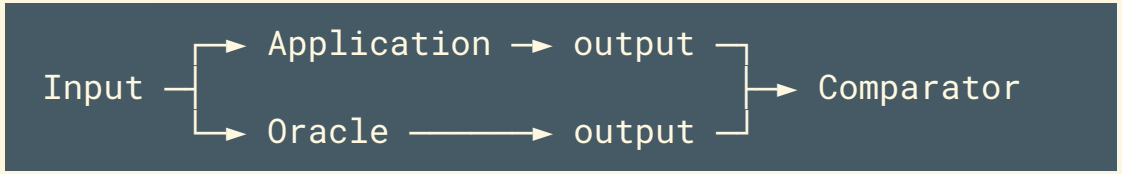
After John Hughes, *How to Specify It!* (2020) · `code/pbt-mat/hypothesis-2/bst_properties.py` · Try: delete the `k == t.key` case in `insert` and see how many properties fail.

A Catalog of Properties

Pattern	Meaning	Example
Invariant	a quantity is preserved or bounded	`sum(shares)==total`, `max-min <= 1`
Round-trip	decode(encode(x)) == x	`parse(format(t)) == t`
Conservation	totals balance system-wide	balances sum to zero
Oracle	agrees with a trusted reference	one-cent-at-a-time split
Idempotence	f(f(x)) == f(x)	`normalize` LLM category text
Metamorphic	related inputs → related outputs	insert order doesn't matter
Robustness	never fails in an undocumented way	parser raises only `ParseError`

Reproducibility: randomness comes from a **seed**. Hypothesis replays past failures from `.hypothesis/`. CI uses `derandomize=True`. Pin a found bug forever with `@example(total=1, n=2)`.

Differential Testing



Run the same input through **several implementations**. Any disagreement means at least one is wrong. You don't need to know the right answer.

Csmith generates random valid C programs, compiles each with gcc, clang, and tcc, and compares the outputs. It found **hundreds** of bugs in GCC and Clang.

```
def split_one_cent_at_a_time(total, n):
    shares = [0] * n
    for i in range(total):
        shares[i % n] += 1      # obviously correct
    return shares

@given(total=st.integers(0, 5_000),
       n=st.integers(1, 50))
def test_split_agrees_with_oracle(total, n):
    assert split_evenly(total, n) == \
           split_one_cent_at_a_time(total, n)
```

Unlike the sum property, this also checks **who** gets the leftover cents. When you optimize, keep the slow version as the oracle.

Contract Tests: Is the Fake Honest?

```python
@pytest.fixture(params=["memory", "sqlite"])
def store(request):
    if request.param == "memory":
        yield Ledger()
    else:
        db = SqliteLedger(":memory:"); yield db; db.close()

def test_add_many_is_atomic(store):
    bad = Transaction(PIZZA.date, "Refund", -100, "alice", ("bob",))
    with pytest.raises((ValueError, sqlite3.IntegrityError)):
        store.add_many([PIZZA, bad])
    assert store.all() == []        # all or nothing
```

> **True story:** the first in-memory `add_many` looped over `add`. SQLite rolled back and stored nothing; the fake stored the pizza. The fake was **more permissive than the real thing**, so tests using it would pass on code that production rejects. Running one suite on both implementations caught it.

Code Coverage: What Did the Tests Never Run?

- **Function:** was each function called?
- **Statement:** did each line run?
- **Branch:** did both sides of every `if` run?
- **Condition:** was each boolean sub-expression both true and false?

```
$ pytest --cov        # branch = true, fail_under = 90
Name           Stmts Miss Branch BrPart Cover Missing
-----------------------------------------------------
app/ledger.py     53    1      6      0   98%  59
app/parser.py     38    3     20      3   90%  75,77,79
app/split.py      19    0      8      0  100%
-----------------------------------------------------
TOTAL            215    4     42      3   97%
```

Coverage is a **flashlight**. It doesn't prove the lit corners are clean, but it shows the dark ones.

- `ledger.py:59` is `add_line`. Nothing calls it, so it's **dead code**: delete it instead of testing it
- `parser.py:75–79` are `format_transaction`'s error paths. The round-trip property only generates **valid** input, so they never run

Statement < Branch < Condition, and Coverage's Limit

```
return [base + 1 if i < leftover else base
        for i in range(n_people)]
```

`split_evenly(900, 3)` gives **100% statement** coverage, but `leftover == 0`, so `base + 1` never runs. Branch coverage reports it.

```
if not text or any(c in text for c in ",;\n"):
```

Branch: `""` and `"Pizza"`. Condition also needs `"Pizza, large"`.

100% coverage ≠ tested. Coverage measures what ran, not what was checked:

```
def test_split_runs():
    split_evenly(1000, 3)  # asserts nothing
```

- Use coverage **negatively**: 0% means untested
- Set a **floor** in CI, not a target (Goodhart's law)

Mutation Testing: Testing the Tests

Make small changes to the **program**. Each changed copy is a **mutant**. Run the suite on it: if a test fails, the mutant is **killed**; if all pass, it **survived**. **Score** = killed / (total − equivalent).

```
if (a && b) { c = 1; }    →    if (a || b) { c = 1; }
else        { c = 0; }         else        { c = 0; }
```

To kill this mutant you need a test where `a ≠ b`. A survivor tells you **exactly which test is missing**.

Mutation operators: delete a statement; duplicate one (`goto fail;`); bool → `true`/`false`; `+` ↔ `-`, `*`; `<` ↔ `<=`; `0` → `1`; remove a method body

Planted one-character bugs are good stand-ins because real code is usually *close* to correct.

Coverage tells you the fire alarm **has power**. Mutation testing is the **fire drill**.

Mutation Testing Finds a Real Gap DEMO

```
$ python -m mutation.mutate --tests tests/test_split.py
...
mutation score: 9/9 killed (100%)

$ python -m mutation.mutate --target app/parser.py \
      --tests tests/test_parser.py
[  0] killed   line  35: NotEq -> Eq
[  1] killed   line  51: LtE -> Lt
[  2] SURVIVED line  76: LtE -> Lt
[  3] killed   line  35: constant 5 -> 6
[  4] SURVIVED line  51: constant 0 -> 1
[  5] SURVIVED line  76: constant 0 -> 1
mutation score: 3/6 killed (50%)
```

Line 51: `if amount_cents <= 0: raise ParseError`

- `<= 0 → < 0` is **killed**, because a test uses 0
- `<= 0 → <= 1` **survives**: the parser now rejects a 1-cent amount and **no test notices**

The suite checks the rejecting side of the boundary (0) but not the accepting side (1). **Coverage of line 51 is 100%.**

Fix: one test that `"...,Gum,1,alice,bob"` parses.

Cost: one full test run per mutant, so run it nightly or only on changed files. Equivalent mutants (`n < 1` → `n <= 0`) need a human to judge.

Fuzz Testing

> **Fuzzing** feeds a program huge amounts of generated random or malformed input (**fuzz**) and watches for **crashes, hangs, and failed assertions**. Started with Barton Miller's 1988 class at **UW-Madison**: random bytes crashed about ⅓ of Unix utilities.

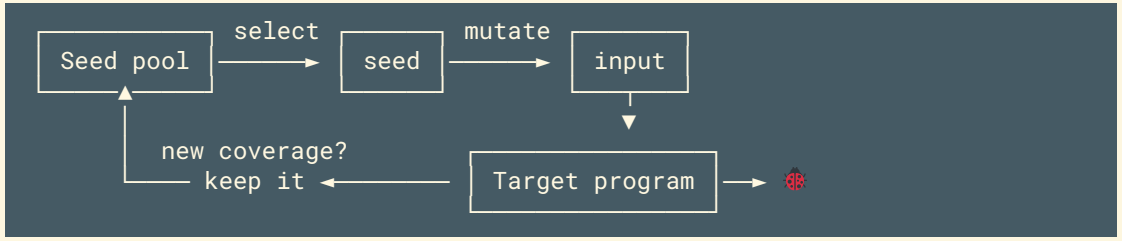
```python
# LLM-style first draft: app/parser_v1.py
def parse_transaction(line):
    f = line.split(",")
    date = datetime.date.fromisoformat(f[0])
    amount = int(f[2])
    parts = tuple(f[4].split(";"))
    return Transaction(date, f[1], amount, f[3], parts)
```

```python
# The harness: the contract
try:    parse(line)
except ParseError: pass          # allowed
except Exception as e: report_crash(line, e)
```

```
$ python -m fuzz.fuzz_parser --target buggy
[1] CRASH IndexError: list index out of range
    input:     '2026-12-01,Lab s'
    minimized: '2026-12-01'
[2] CRASH ValueError: Invalid isoformat ...
    input:     '2026-1--31,Lab snacks,105,...'
    minimized: ''
50000 iterations, 2 distinct crash(es)
```

"Nobody types that" isn't true: **attackers do**, and so do truncated uploads and LLMs writing tool arguments.

Coverage-Guided Fuzzing




```
seed "2026-09-12,Pizza night,2350,alice,alice;bob"
  ├─ "2026-09-12,Pizza night"   field-count  → KEEP
  ├─ "2026-09-1x,Pizza,2350,…"  bad date     → KEEP
  ├─ "2026-09-12,Pizza,23x0,…"  bad amount   → KEEP
  └─ "2026-09-12,Pizzq night…"  nothing new  → drop
```

Blind fuzzing doesn't learn. **Coverage feedback** keeps inputs that reach **new paths**, so coverage becomes a live search signal (AFL++, libFuzzer, Atheris).

	PBT	Fuzzing
Property	rich, domain	"no crash"
Duration	~100 examples	millions, hours
Guidance	generator	coverage

Code with a **spec** gets properties. Code that reads **untrusted input** gets fuzzed too.

Static Analysis: Checking Without Running

```python
def split_evenly(total_cents: int, n_people: int) -> list[int]:
    share = total_cents / n_people     # float!
    return [share] * n_people
```

```
$ mypy --strict app
app/split.py:3: error: List item 0 has incompatible
    type "float"; expected "int"
```

```
$ ruff check .
B905 `zip()` without an explicit `strict=` parameter
B017 `pytest.raises(Exception)` is too broad
```

- **Type checker:** turns a policy ("money is integer cents") into a mechanical check
- **Linter:** `zip` silently stops at the shorter list, so someone could be charged nothing without any error

> "It type-checks, so it works" is wrong. `total - share` and `total + share` have the same type. Static analysis is the **cheapest first filter**, not a replacement for tests.

Continuous Integration: Every Check, Every Change

```yaml
jobs:
  lint:       # cheapest first: fail in seconds
    steps: [ ruff check . ]
  typecheck:
    steps: [ mypy app ]
  test:
    needs: [lint, typecheck]
    strategy:
      matrix: { python: ["3.11", "3.12", "3.13"] }
    env: { HYPOTHESIS_PROFILE: ci }  # derandomized
    steps: [ pytest --cov ]          # floor: 90%
  fuzz-smoke:
    needs: [test]
    steps: [ python -m fuzz.fuzz_parser --seed 0 ]
```

- **Cheapest first**, so feedback is fast
- **Deterministic:** fixed seeds, so a random failure doesn't block the whole team
- **Nightly** pipeline for slow work: long fuzzing, 20k Hypothesis examples, mutation testing
- PRs include tests; a bug fix includes a test that **fails without the fix**
- **Smoke tests** run against the built container and again after deploy *(Lecture 19)*

> CI green ≠ correct. It certifies exactly the checks **you wrote**.

Abbreviated from slides/code/terpledger/.github/workflows/ci.yml

LLM-Generated Tests: The Oracle Problem

When the model reads the code to write the test, the **code becomes the oracle**. Ask it to test the buggy `[total // n] * n`:

```
def test_split_evenly_uneven():
    assert split_evenly(1000, 3) == [333, 333, 333]   # "expected" = what the code returned
```

> The bug is now **enshrined**. When someone later fixes it correctly, this test reports the fix as a regression. A test that asserts the code does what it does protects nothing.

Where LLMs help:

- "Here's the **docstring**. List 15 edge cases." (leap days, Unicode, commas in descriptions…)
- "What **properties** should `split_evenly` satisfy?" (sometimes a wrong one: "all shares equal")
- Explaining failures, and writing fixtures and other boilerplate

Grading an AI-Written Suite with Mutation Testing

```python
# weak_tests/test_split_ai_generated.py
def test_split_evenly_returns_list():
    result = split_evenly(1000, 3)
    assert isinstance(result, list)
    assert len(result) == 3

def test_split_evenly_invalid_people():
    with pytest.raises(Exception):
        split_evenly(100, 0)

def test_balances_returns_dict():
    result = balances([t])
    assert "alice" in result and "bob" in result
```

```
$ pytest weak_tests --cov=app.split --cov-branch
app/split.py   19   0   8   0   100%

$ python -m mutation.mutate \
    --tests weak_tests/test_split_ai_generated.py
[  2] SURVIVED line 44: Add -> Sub   # payer OWES
[  6] SURVIVED line 30: Lt -> LtE    # leftover rule
...
mutation score: 2/9 killed (22%)
```

Same 100% coverage: real suite **9/9**, AI suite **2/9**. Mutation testing tells them apart in seconds.

A Workflow for AI-Assisted Testing

1. **Prompt with the specification**, not just the code. Ask for boundaries, specific exceptions, and properties
2. **Ask for ideas before code**: edge cases and properties, which **you** filter
3. **Review every assertion**: did this expected value come from the **requirements** or from **running the code**?
4. **Watch for test smells**: `raises(Exception)`, assertions that only check type, length, or key presence, tests that re-implement the function, heavy mocking
5. **Grade mechanically**: coverage shows what was skipped, mutation shows what was never checked. Feed survivors back: *"changing* `+=` *to* `-=` *on line 44 survives; write a test that kills it."*
6. **Make every new test fail once** by breaking the code by hand

Which Technique Catches Which Bug?

The mistake	Caught by
`[total // n] * n` loses leftover cents	unit test (non-divisible case); property (always)
negative withdrawal becomes a deposit	**test design** exposed a missing requirement
triangle checks only one inequality	partitioning: permute the longest side
buggy `reverse` does nothing	unit test, or a **stronger** property
leftover cents go to the wrong person	differential test vs. one-cent oracle
in-memory fake accepts what SQLite rejects	contract test over both backends
`format_transaction` error paths never run	coverage report
no test checks that 1 cent is accepted	**mutation testing only** (line coverage 100%)
parser `IndexError` on `"2026-12-01"`	fuzzing; robustness property
`total / n` makes money a float	type checker, instantly
AI suite asserts only types and lengths	mutation testing (22% at 100% coverage)
any of the above, reintroduced next month	**CI**

> There's almost no overlap. This is **defense in depth**: independent layers with independent blind spots.

Key Takeaways

- Testing shows the **presence** of bugs, never their absence. Bugs cost more the later you find them
- Most tests should be **small, fast unit tests**. Regression testing re-runs all of them on every change
- Choose inputs **systematically**: one per partition, **both** sides of each boundary, negative cases, a clear **oracle**. Writing tests exposes missing requirements
- `pytest`: assertions, parametrize, fixtures. Use **test doubles** for slow or random dependencies, preferring fakes to mocks. **TDD**: red → green → refactor
- **Properties** state laws over whole input domains, and **shrinking** gives minimal counterexamples. A weak property ("sorted", "reverse twice") lets bugs through
- **Coverage** shows what never ran. **Mutation testing** shows what was never *checked*. **Fuzz** anything that reads untrusted input
- **CI** runs every check on every change. LLMs are useful test assistants, but you check every expected value against the **spec**

Try it: `cd slides/code/terpledger && pytest --cov && python -m mutation.mutate && python -m fuzz.fuzz_parser --target buggy`