Prof. Aviv Zohar

Semester A, 2026–7

What tests cost, which to keep

Brittle tests, mutants and coverage



A test suite is code you maintain

todo/app.py — the program

```
3  MAX_TITLE = 80
   ⋮
6  def add_item(items, text):
7      if len(text) > MAX_TITLE:
8          raise ValueError(text)
```

tests/test_app.py — its tests

```
21 def test_longest_title(items):
22     add_item(items, "x" * 80)
   ⋮
25 def test_too_long(items):
26     with pytest.raises(ValueError):
27         add_item(items, "x" * 81)
```

Both get read, refactored and repaired.

A test suite is code you maintain

New rule: titles up to 60 characters (was 80).

todo/app.py — the program

```
- MAX_TITLE = 80
+ MAX_TITLE = 60
⋮
6  def add_item(items, text):
7      if len(text) > MAX_TITLE:
8          raise ValueError(text)
```

tests/test_app.py — its tests

```
21 def test_longest_title(items):
22     add_item(items, "x" * 80)
⋮
25 def test_too_long(items):
26     with pytest.raises(ValueError):
27         add_item(items, "x" * 81)
```

The change you meant to make.

A test suite is code you maintain

New rule: titles up to 60 characters (was 80).

todo/app.py — the program

```
-  MAX_TITLE = 80
+  MAX_TITLE = 60
   ⋮
 6 def add_item(items, text):
 7     if len(text) > MAX_TITLE:
 8         raise ValueError(text)
```

tests/test_app.py — its tests

```
21 def test_longest_title(items):
-      add_item(items, "x" * 80)
+      add_item(items, "x" * 60)
   ⋮
25 def test_too_long(items):
26     with pytest.raises(ValueError):
-          add_item(items, "x" * 81)
+          add_item(items, "x" * 61)
```

Two tests described the old limit.

A test suite is code you maintain

New rule: titles up to 60 characters (was 80).

todo/app.py — the program

```
- MAX_TITLE = 80
+ MAX_TITLE = 60
  ⋮
6 def add_item(items, text):
7     if len(text) > MAX_TITLE:
8         raise ValueError(text)
```

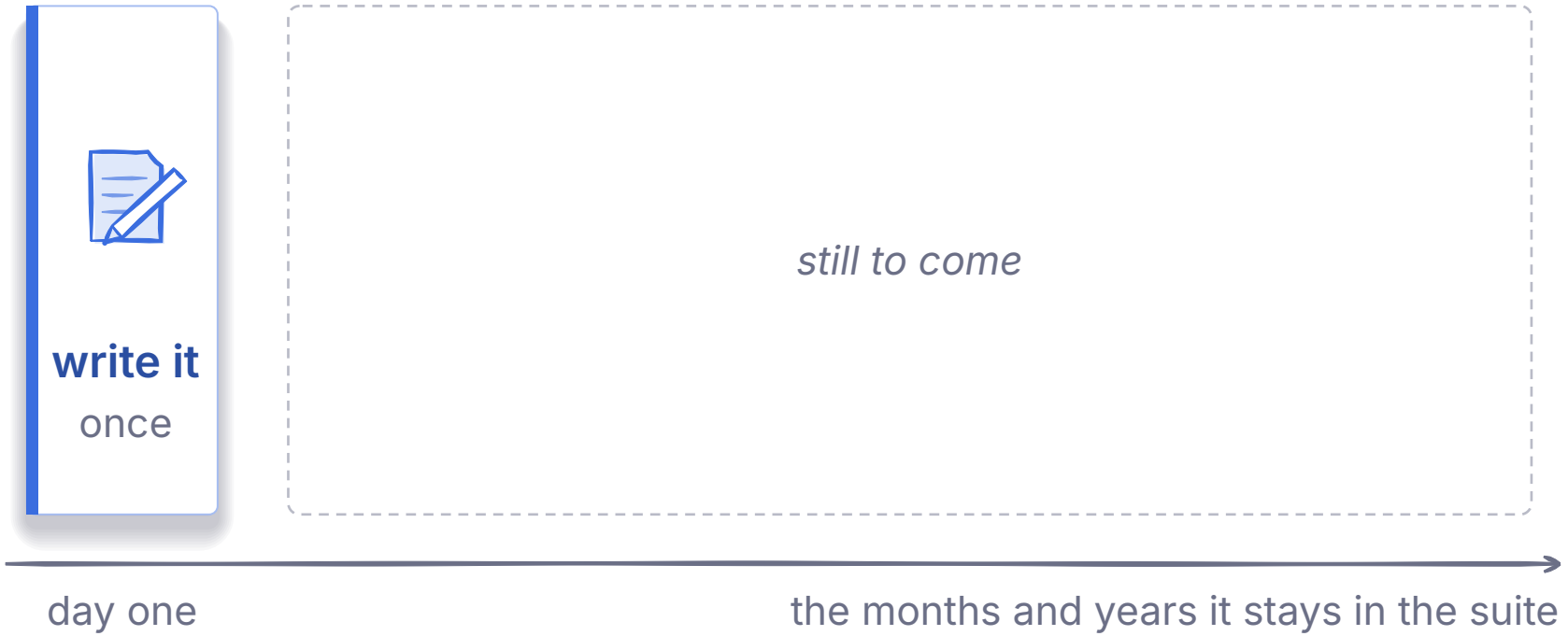
tests/test_app.py — its tests

```
21 def test_longest_title(items):
-      add_item(items, "x" * 80)
+      add_item(items, "x" * 60)
   ⋮
25 def test_too_long(items):
26     with pytest.raises(ValueError):
-          add_item(items, "x" * 81)
+          add_item(items, "x" * 61)
```

One change: estimate, update and review both.

Most of a test's cost comes after it is written

On the day you write it, that is the whole cost.



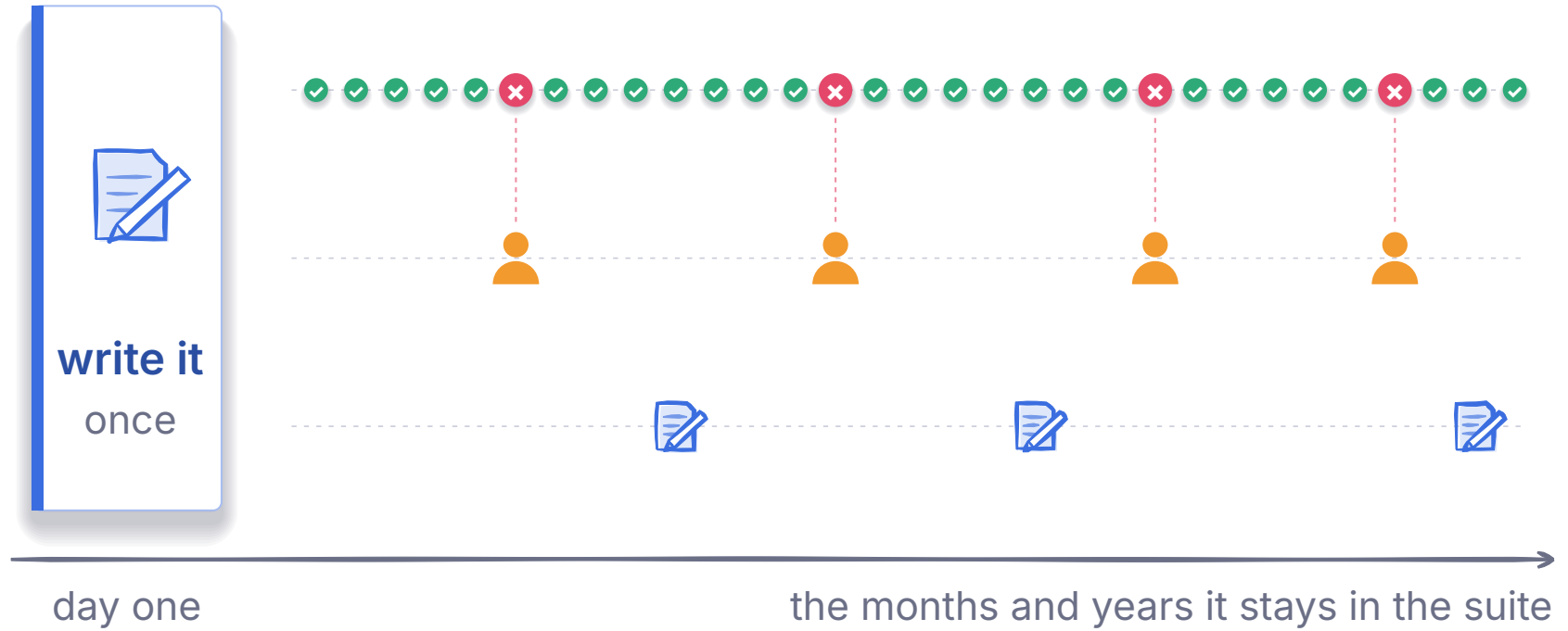
Most of a test's cost comes after it is written

Adding a test means agreeing to maintain it.

runs on every change

read when it fails

updated when the behaviour changes



illustrative timeline

A change-detector test checks the code's arrangement

tests/test_cart.py

```
1 def test_add():
2     cart = Cart()
3     cart.add("book", 12)
4     assert cart._items == [("book", 12)]
```

what a user sees

book	12
total	**12**

The underscore in `_items` marks it as the cart's own.

cart.py — inside the cart

```
3 class Cart:
4     def __init__(self):
5         self._items = []
```

A change-detector test checks the code's arrangement

tests/test_cart.py

```python
def test_add():
    cart = Cart()
    cart.add("book", 12)
    assert cart._items == [("book", 12)]
```

what a user sees

book	12
total	**12**

Same book, same total.

cart.py — inside the cart

```python
class Cart:
    def __init__(self):
-       self._items = []
+       self._items = {}
```

A list becomes a dictionary.

A change-detector test checks the code's arrangement

tests/test_cart.py — 1 failed

```
1 def test_add():
2     cart = Cart()
3     cart.add("book", 12)
4     assert cart._items == [("book", 12)]
```

what a user sees

book	12
total	**12**

Red anyway: it was checking that the items sit in a list.

cart.py — inside the cart

```
3 class Cart:
4     def __init__(self):
-         self._items = []
+         self._items = {}
```

A change-detector test checks the code's arrangement

tests/test_cart.py — 1 failed

```python
def test_add():
    cart = Cart()
    cart.add("book", 12)
    assert cart._items == [("book", 12)]
```

cart.py — inside the cart

```
class Cart:
    def __init__(self):
-       self._items = []
+       self._items = {}
```

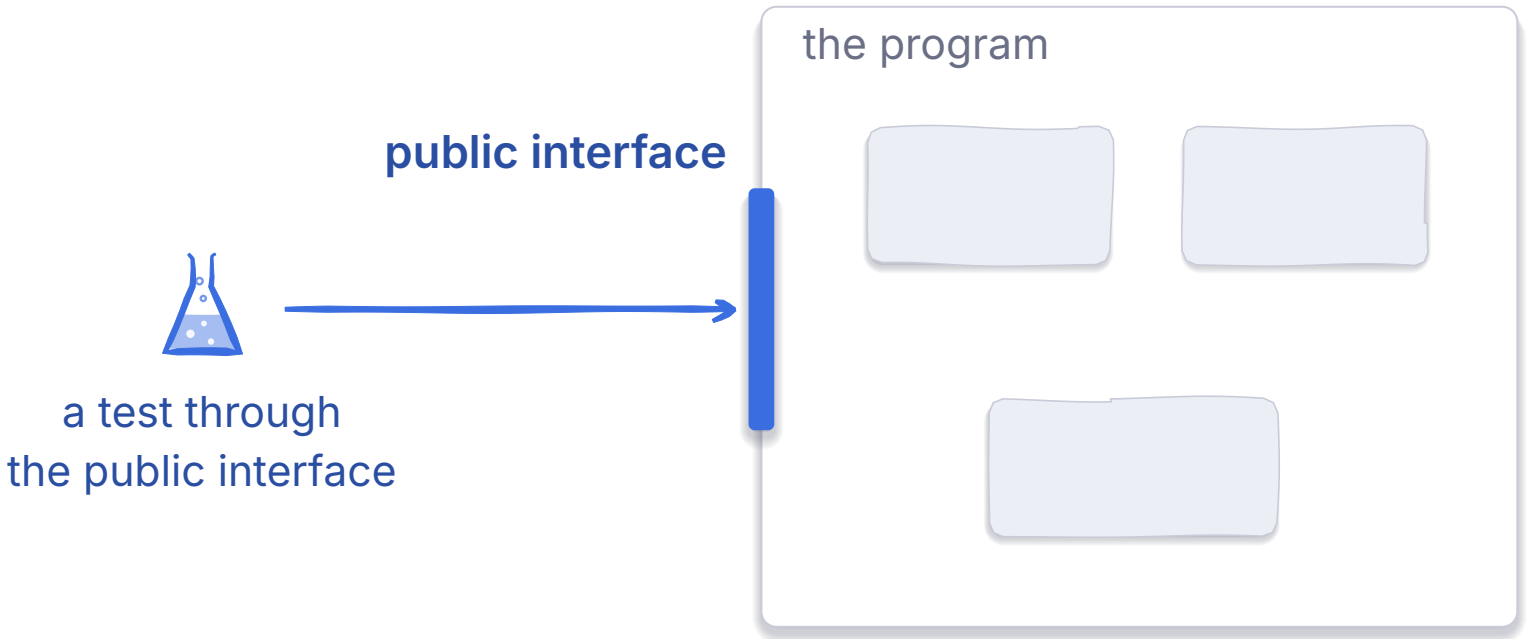
what a user sees

book	12
total	**12**

Negative value: a test to fix, and nothing broke.

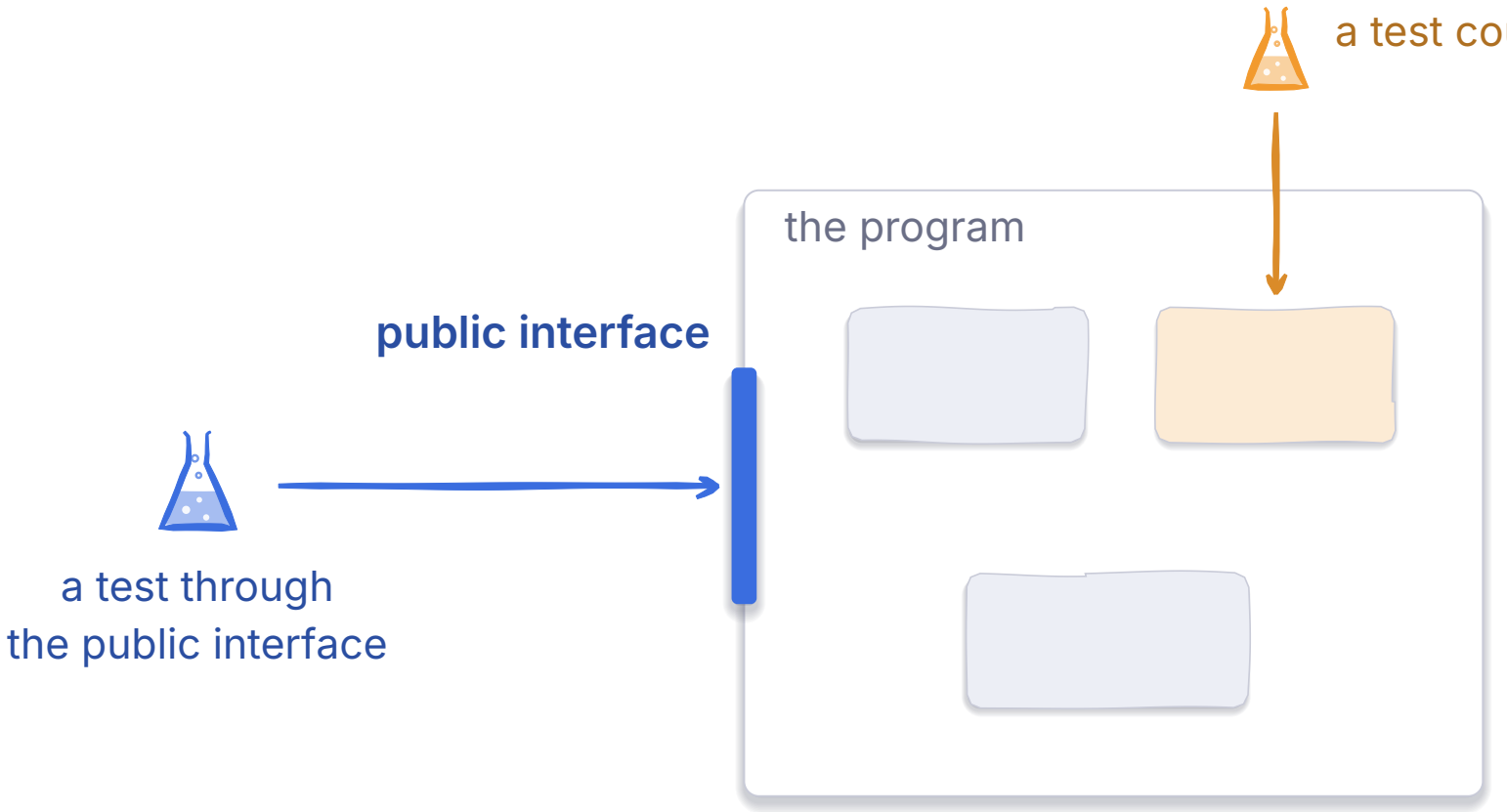
Google Testing Blog, "Change-Detector Tests Considered Harmful", 2015

Couple tests to behaviour, not to structure



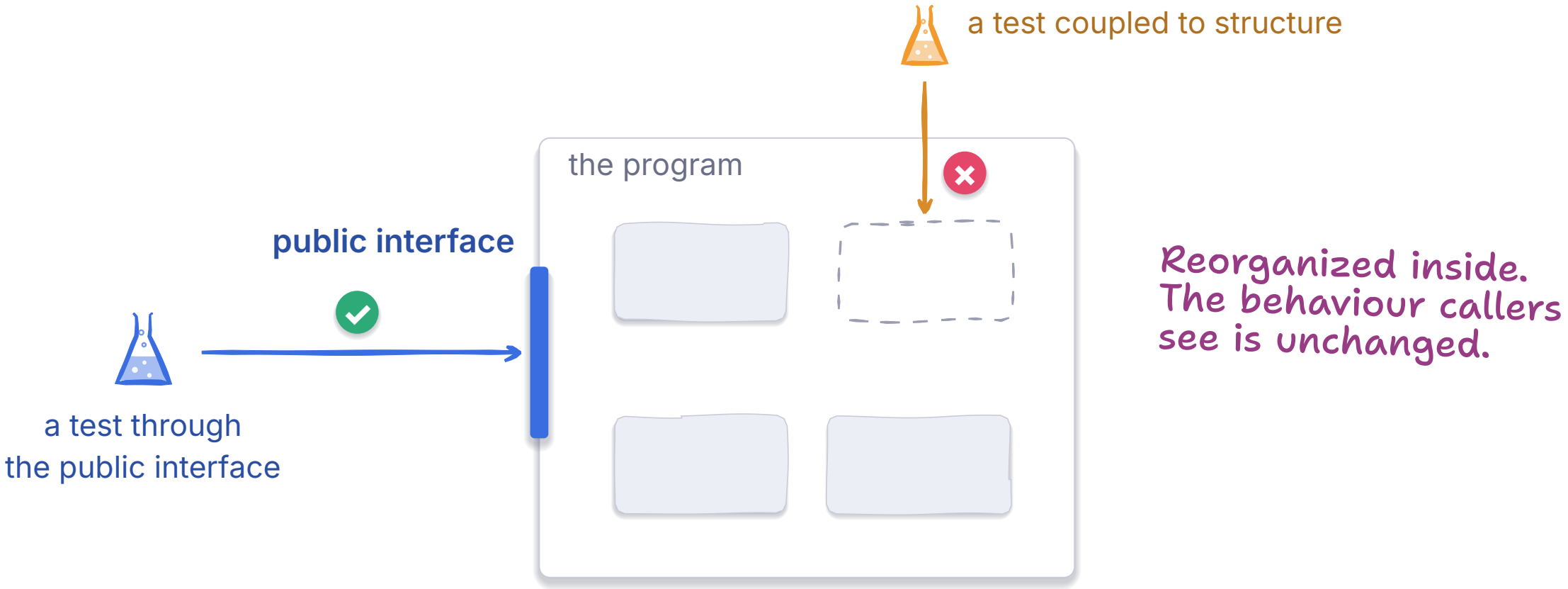
The public interface: the functions or module that callers use. A test can call it without opening a browser.

Couple tests to behaviour, not to structure



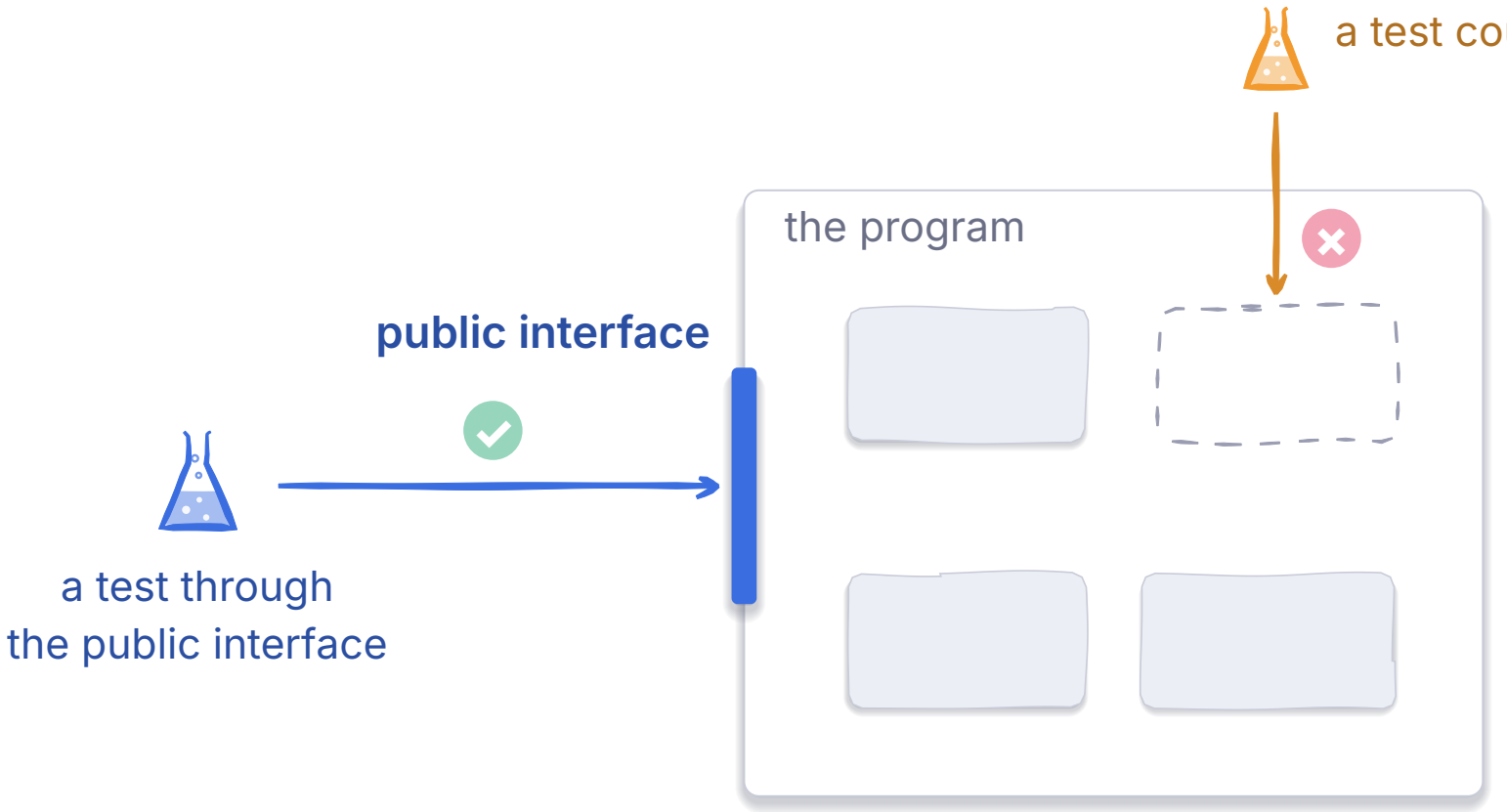
It depends on one particular arrangement inside.

Couple tests to behaviour, not to structure



Reorganized inside.
The behaviour callers
see is unchanged.

Couple tests to behaviour, not to structure



Couple to behaviour, decouple from structure.

Kent Beck

Kent Beck, 2019 · Software Engineering at Google, ch. 12

The cart test, coupled to behaviour

before — fails on a refactor

```
1 def test_add():
2     cart = Cart()
3     cart.add("book", 12)
4     assert cart._items == [("book", 12)]
```

Reaches into the cart's own list.

after — passes either way

```
1 def test_add():
2     cart = Cart()
3     cart.add("book", 12)
4     assert cart.items() == [("book", 12)]
5     assert cart.total() == 12
```

Calls the cart's public methods. How they store the items stays inside the cart.

A flaky test can fail with no change to the code

```shell
~/todo $ uv run pytest -q
42 passed in 3.1s
```

Same code, run again.

```shell
~/todo $ uv run pytest -q
1 failed, 41 passed in 3.4s
```

> of test runs reported a flaky result
>
> **1.5%**

> of tests showed some flakiness
>
> **16%**

Google Testing Blog, “Flaky Tests at Google and How We Mitigate Them”, 2016

A flaky test can fail with no change to the code

shell

```
~/todo $ uv run pytest -q
42 passed in 3.1s
```

Same code, run again.

shell

```
~/todo $ uv run pytest -q
1 failed, 41 passed in 3.4s
```

of test runs reported a flaky result

1.5%

of tests showed some flakiness

16%

of pass-to-fail transitions involved a flaky test

84%

Google Testing Blog, “Flaky Tests at Google and How We Mitigate Them”, 2016

A flaky test can fail with no change to the code

shell

```
~/todo $ uv run pytest -q
42 passed in 3.1s
```

Same code, run again.

shell

```
~/todo $ uv run pytest -q
1 failed, 41 passed in 3.4s
```

of test runs reported a flaky result

1.5%

of tests showed some flakiness

16%

of pass-to-fail transitions involved a flaky test

84%

Rerun until it is green, and nobody asks why it failed. Tests that fail at random undermine trust in the whole suite.

Google Testing Blog, “Flaky Tests at Google and How We Mitigate Them”, 2016

Repair the cause of the flakiness

the current time — pass in a fixed clock

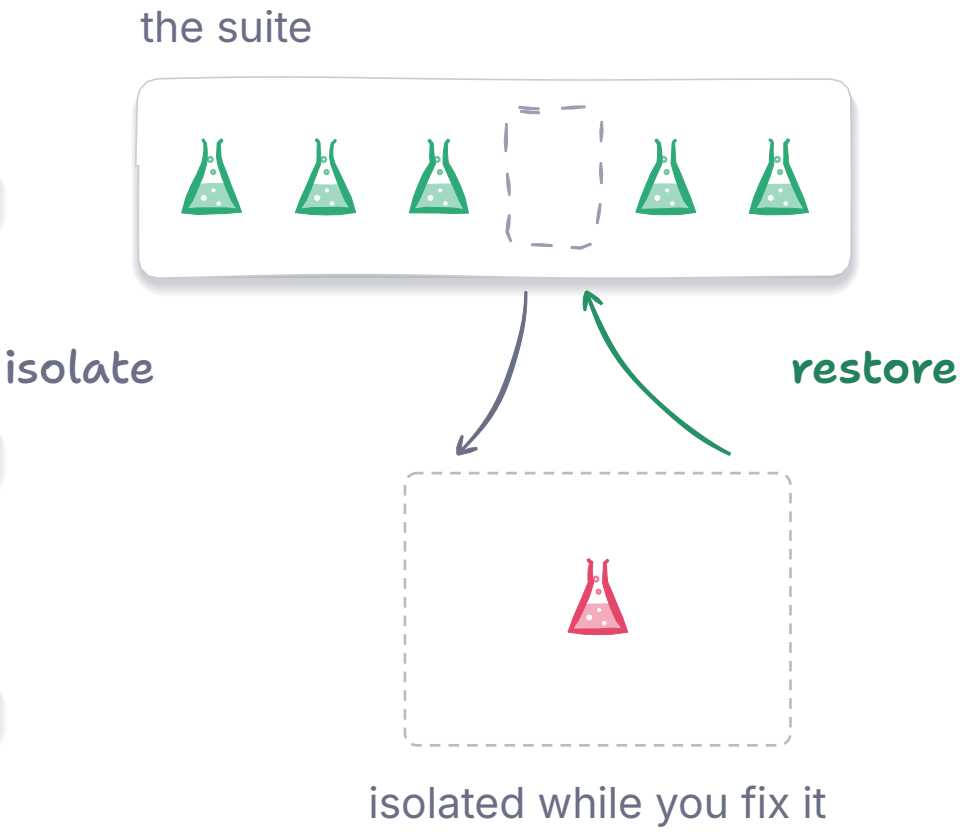
```
- due = date.today()
+ due = clock.today()
```

random input — use a fixed seed

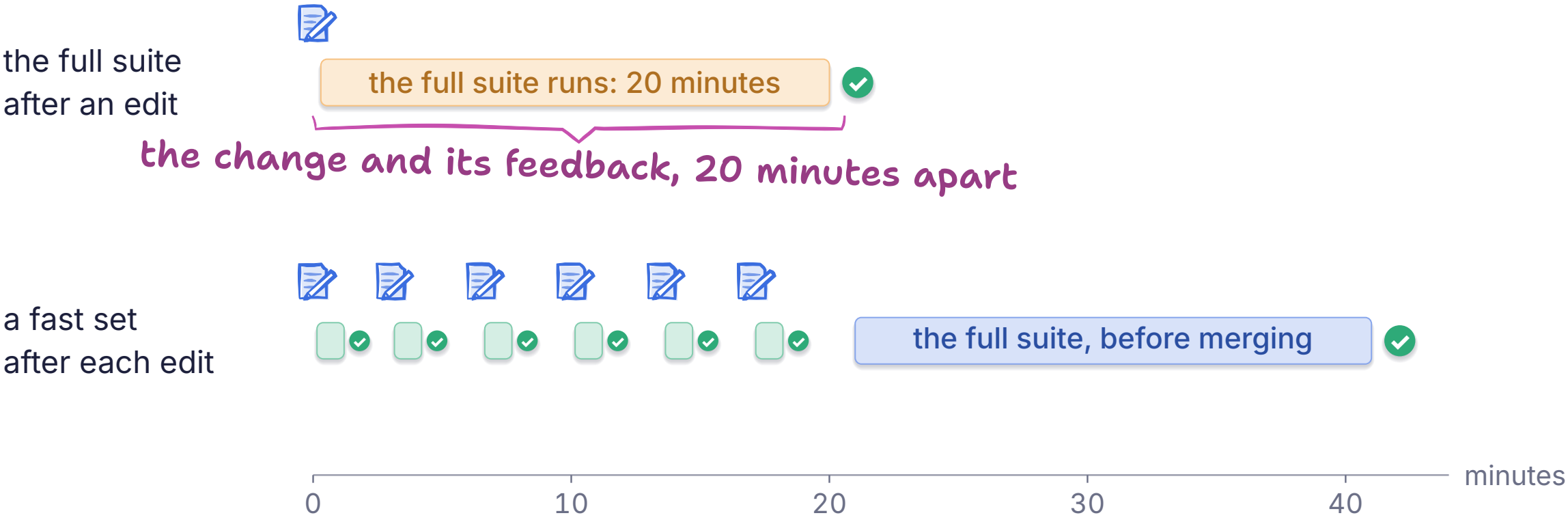
```
- pick = random.choice(names)
+ pick = random.Random(42).choice(names)
```

a fixed delay — wait for the condition

```
- time.sleep(5)
+ expect(rows).to_have_count(1)
```



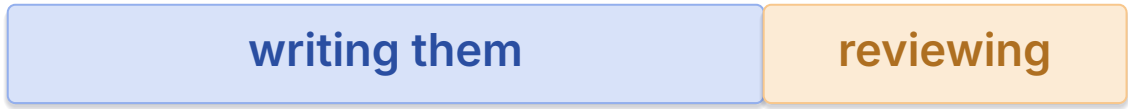
Run a fast set after each edit, the full suite less often



illustrative times

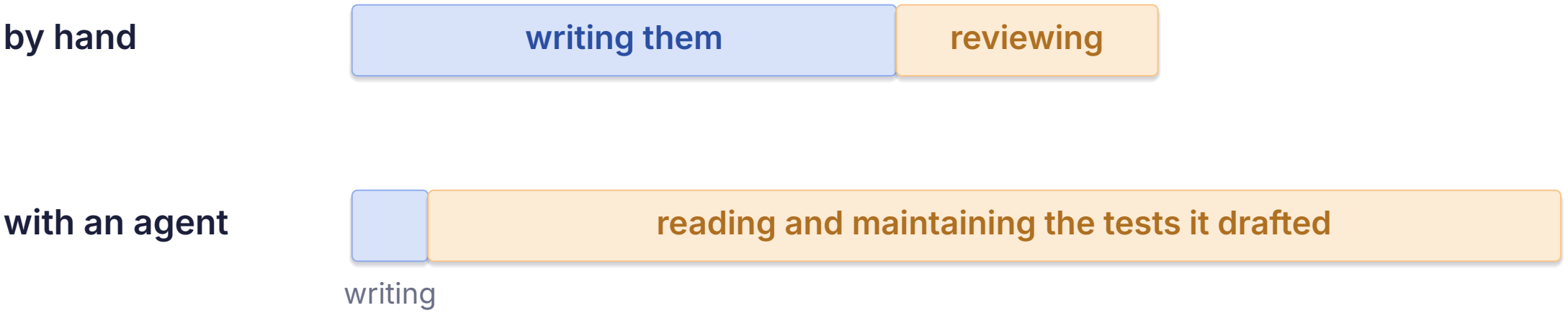
The work moves from writing tests to reviewing them

by hand



illustrative effort

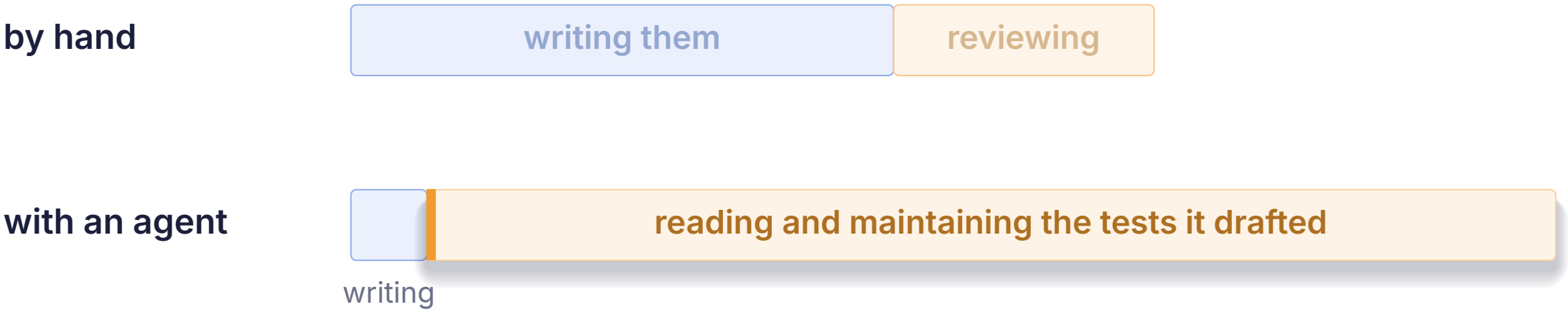
The work moves from writing tests to reviewing them



Perhaps 400 tests before lunch. Read them anyway: a test can pass while checking the wrong thing, like the cart test.

illustrative effort

The work moves from writing tests to reviewing them

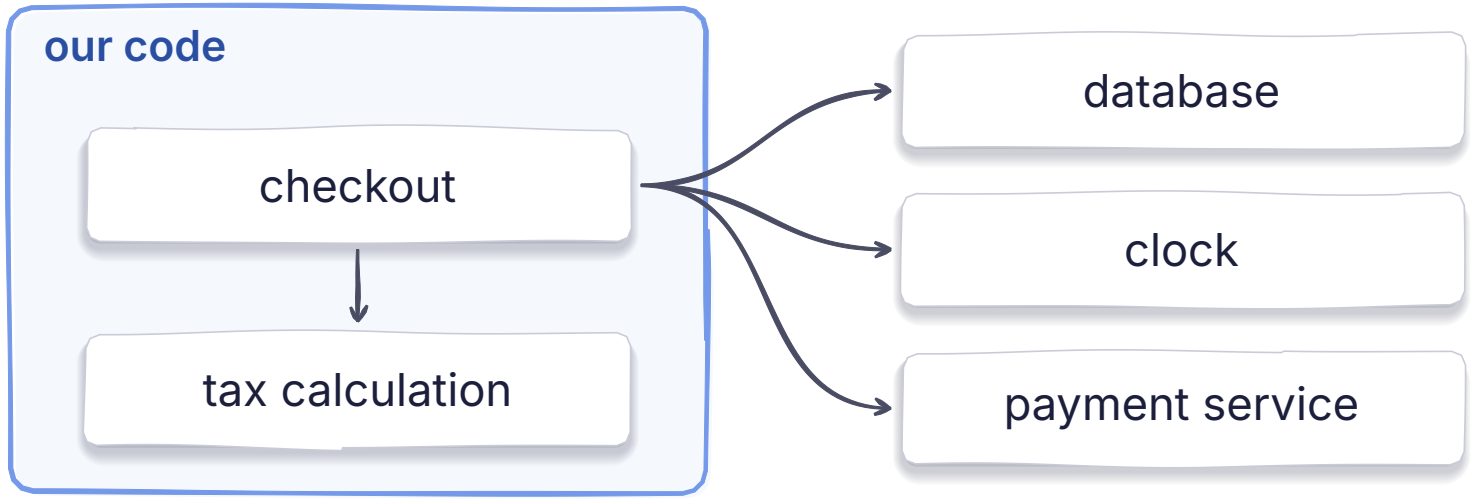


Ask for tests in groups small enough to review. Read each new test, and look closely at any change to a test you already rely on.

How many tests were generated does not measure what you gained.

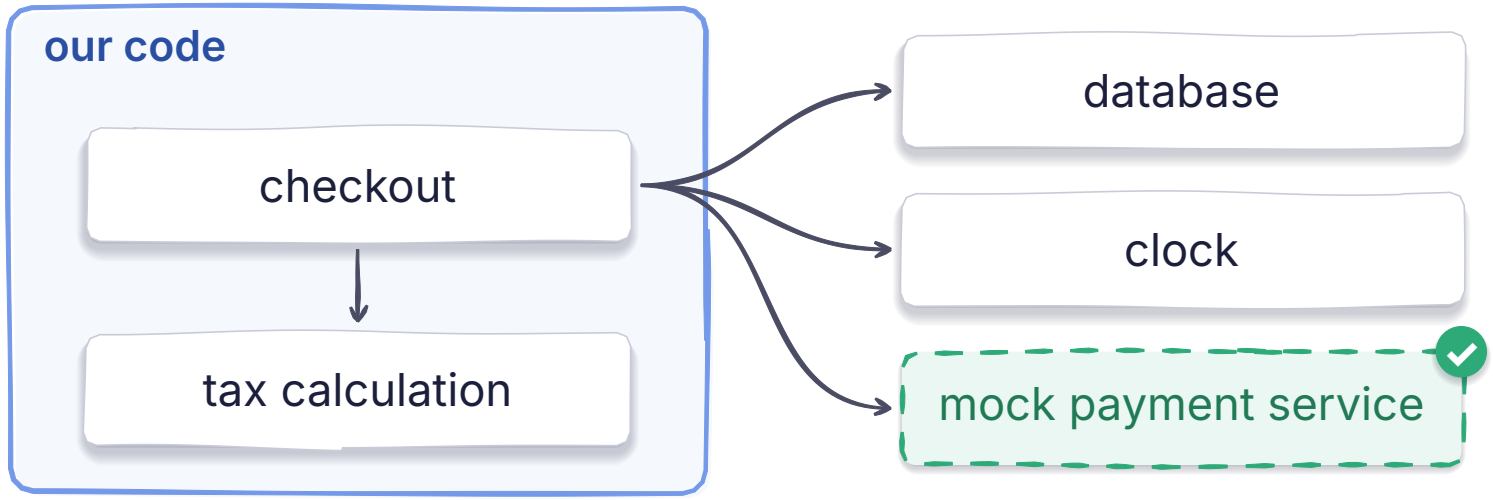
illustrative effort

Replace dependencies, not the code you are testing



Real code talks to things outside itself.

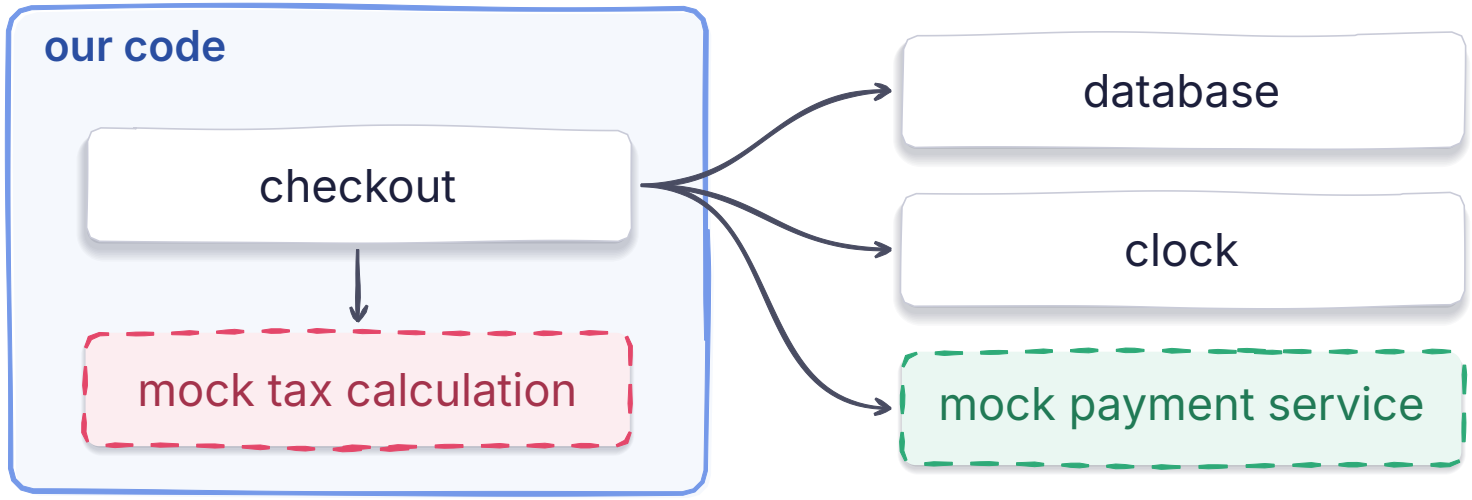
Replace dependencies, not the code you are testing



A mock is a stand-in the test passes in place of the real object. It answers however the test tells it to.

A fake payment service: no real payment is made. A useful mock.

Replace dependencies, not the code you are testing

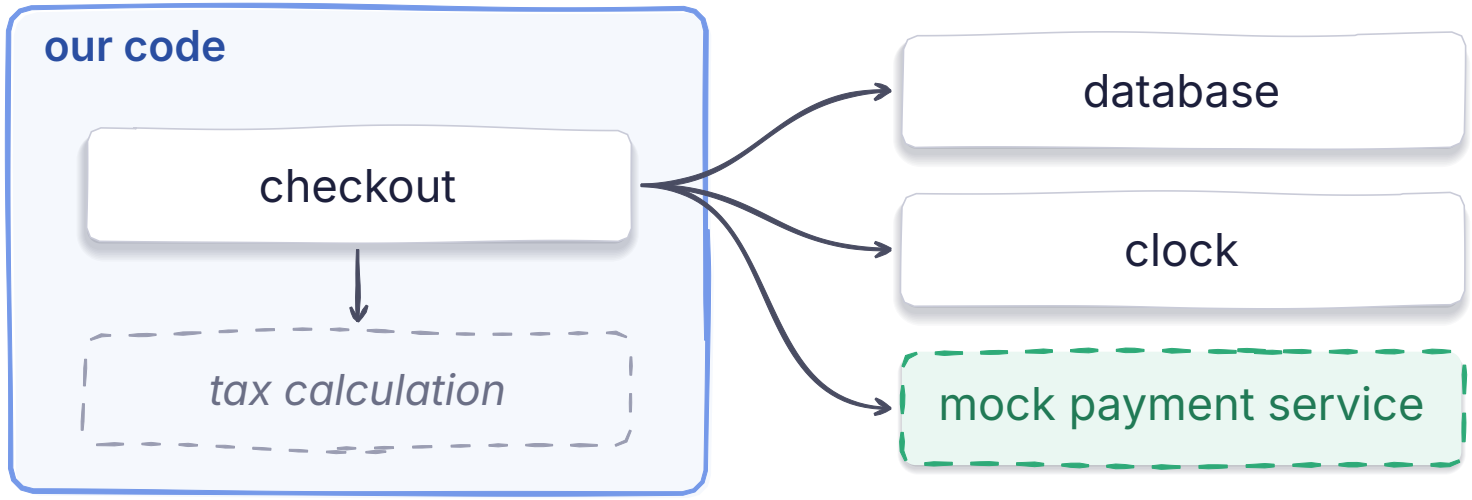


Our own tax calculation, replaced by a mock that returns `118.0`.

tests/test_checkout.py

```
1 def test_total_includes_tax():
2     tax = Mock()
3     tax.apply.return_value = 118.0
4     assert Checkout(tax).total(cart) == 118.0
```

Replace dependencies, not the code you are testing



The assertion checks the number the test configured. The real tax calculation never runs.

tests/test_checkout.py

```
1 def test_total_includes_tax():
2     tax = Mock()
3     tax.apply.return_value = 118.0
4     assert Checkout(tax).total(cart) == 118.0
```

Remove a rule to see whether a test notices

todo/app.py

```
10 def can_add(items, text):
11     text = text.strip()
12     if text == "": return False
13     return all(i.text != text for i in items)
```

rule in place

42 passed

It rejects an item that is empty after trimming.
All 42 tests pass.

Remove a rule to see whether a test notices

todo/app.py

```
10 def can_add(items, text):
11     text = text.strip()
-      if text == "": return False
13     return all(i.text != text for i in items)
```

rule in place

42 passed

Removed for now: a blank item is accepted, so a test should fail.

Remove a rule to see whether a test notices

todo/app.py

```
10 def can_add(items, text):
11     text = text.strip()
-      if text == "": return False
13     return all(i.text != text for i in items)
```

rule in place

42 passed

rule removed

42 passed

No test tried
to add a blank item.

Remove a rule to see whether a test notices

todo/app.py

```
10 def can_add(items, text):
11     text = text.strip()
-      if text == "": return False
13     return all(i.text != text for i in items)
```

tests/test_app.py

```
30 def test_rejects_blank_item():
31     assert can_add([], "   ") is False
```

rule in place

✓ 42 passed

rule removed

✓ 42 passed

test added, rule still removed

✗ 1 failed, 42 passed

It fails while the rule is removed: the failure we wanted.

Remove a rule to see whether a test notices

todo/app.py

```
10 def can_add(items, text):
11     text = text.strip()
12     if text == "": return False
13     return all(i.text != text for i in items)
```

tests/test_app.py

```
30 def test_rejects_blank_item():
31     assert can_add([], "   ") is False
```

rule in place

42 passed

rule removed

42 passed

test added, rule still removed

1 failed, 42 passed

rule restored

43 passed

Fails without the rule, passes with it: the red step of test-driven development, after the fact. Ask for it when the agent writes tests after the code.

illustrative runs

Mutation testing makes the small changes for you

todo/app.py — the original

```
10 def can_add(items, text):
11     text = text.strip()
12     if text == "": return False
13     return all(i.text != text for i in items)
```

One small change at a time. The whole suite runs against each changed version.

mutant 1

```
12     if text != "": return False
```

mutant 2

```
12     if text == "": return True
```

mutant 3

```
13     return True
```

mutant 4

```
12     if text <= "": return False
```

Tools that do this: mutmut for Python, Stryker for JavaScript and TypeScript

Mutation testing makes the small changes for you

todo/app.py — the original

```
10 def can_add(items, text):
11     text = text.strip()
12     if text == "": return False
13     return all(i.text != text for i in items)
```

killed — a test failed

survived — all passed

Have the agent run it on its changes. You inspect the survivors.

mutant 1 — killed

```
12     if text != "": return False
```

mutant 2 — killed

```
12     if text == "": return True
```

mutant 3 — survived

```
13     return True
```

mutant 4 — survived

```
12     if text <= "": return False
```

Tools that do this: mutmut for Python, Stryker for JavaScript and TypeScript

At Google, mutation testing runs on changed code

developers used it
24,000+

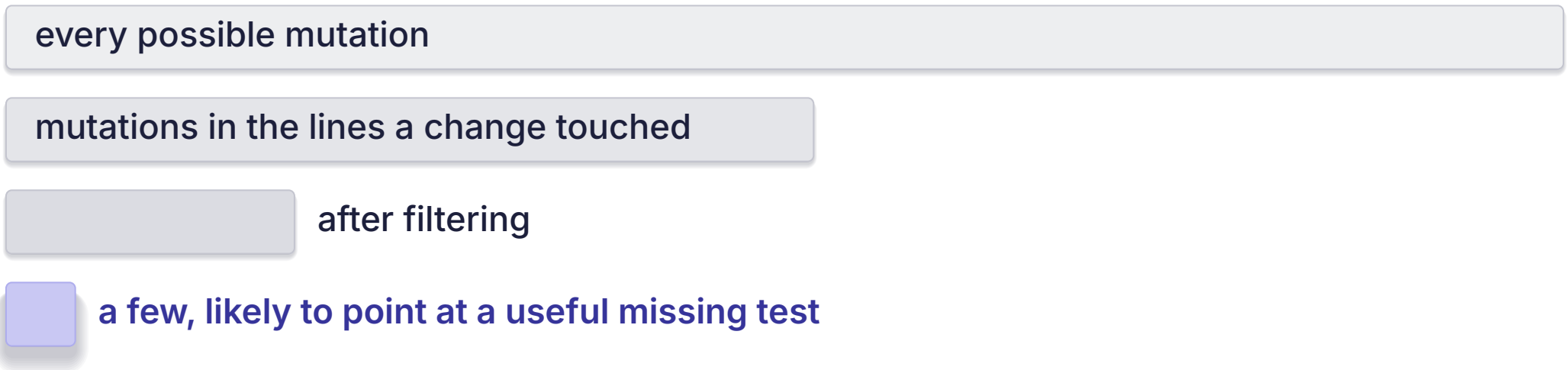
projects
1,000+

Petrović et al., Practical Mutation Testing at Scale, 2021

At Google, mutation testing runs on changed code

developers used it
24,000+

projects
1,000+



Petrović et al., Practical Mutation Testing at Scale, 2021 · bars illustrative

A test that copies the code agrees with its bug

cart.py — the tax is missing

```
5  def total(items):
6      return sum(i.price for i in items)
```

tests/test_cart.py

```
12 def test_total():
13     assert total(cart) == sum(i.price for i in cart)
```

Requirement: the total includes 18% tax.

The expected answer uses the same calculation as the function. This is a tautological test.

A test that copies the code agrees with its bug

cart.py — the tax is missing

```
5  def total(items):
6      return sum(i.price for i in items)
```

tests/test_cart.py

```
12 def test_total():
13     assert total(cart) == sum(i.price for i in cart)
```

Requirement: the total includes 18% tax.

It leaves out the tax exactly like the function does.

- ✓ passes with the bug
- ✗ fails once the tax is added

A test that copies the code agrees with its bug

cart.py — the tax is missing

```
5  def total(items):
6      return sum(i.price for i in items)
```

tests/test_cart.py

```
12 def test_total():
13     assert total(cart) == sum(i.price for i in cart)
```

tests/test_cart.py — assert 100 == 118.0

```
16 def test_total_adds_tax():
17     cart = [Item(price=50), Item(price=50)]
18     assert total(cart) == 118.0
```

Requirement: the total includes 18% tax.

50 + 50 = 100,
plus 18% tax: 118

The expectation comes from the requirement. It fails now and points at the missing tax.

Check the assertion against the test's name

tests/test_app.py

```
20 def test_clear_completed_removes_done_items():                    (a)
21     items = list_with(done=["Buy milk"], open=["Call mom"])
22     cleared = clear_completed(items)
23     assert cleared is not None                                    (b)
```

(a) The name promises that done items are removed.

(b) The assertion only checks that some result came back. The test passes even if nothing is removed.

Check the assertion against the test's name

tests/test_app.py

```
20 def test_clear_completed_removes_done_items():
21     items = list_with(done=["Buy milk"], open=["Call mom"])
22     cleared = clear_completed(items)
-      assert cleared is not None
+      assert visible_items(cleared, filter="all") == ["Call mom"]
```

start
action

expected

Now the assertion checks the behaviour in the name: only "Call mom" is left after clearing.

Line coverage shows what ran, not what was checked

initials.py

```
1 def initials(name):
2     words = name.split()
3     letters = [w[0].upper() for w in words]
4     return "".join(letters)
```

Which lines ran,
not what was checked.

A line that never ran
was not tested at all.

Line coverage shows what ran, not what was checked

initials.py — green: ran during the suite

```python
def initials(name):
    words = name.split()
    letters = [w[0].upper() for w in words]
    return "".join(letters)
```

tests/test_initials.py — 1 passed

```python
def test_initials():
    print(initials("Alan Turing"))
```

line coverage of initials.py

100%

The test prints the result. There is no assertion, and every line runs.

Line coverage shows what ran, not what was checked

initials.py — green: ran during the suite

```
1  def initials(name):
2      words = name.split()
-      letters = [w[0].upper() for w in words]
+      letters = [w[0].lower() for w in words]
4      return "".join(letters)
```

tests/test_initials.py — 1 passed

```
1  def test_initials():
2      print(initials("Alan Turing"))
```

line coverage of initials.py

100%

Now it returns "at". Still 100%, still passing: nothing compares the result with "AT".

Line coverage shows what ran, not what was checked

initials.py — green: ran during the suite

```
1  def initials(name):
2      words = name.split()
-      letters = [w[0].upper() for w in words]
+      letters = [w[0].lower() for w in words]
4      return "".join(letters)
```

tests/test_initials.py — 1 passed

```
1  def test_initials():
2      print(initials("Alan Turing"))
```

line coverage of initials.py

100%

Low coverage points at untested code. **High coverage** shows that many lines ran: read what the tests check.

pytest-cov lists the lines no test ran

```shell
~/todo $ uv add --dev pytest-cov
~/todo $ uv run pytest --cov=. --cov-report=term-missing
```

`--dev`: add the plugin as a development dependency
`--cov=.`: measure coverage for the project
`--cov-report=term-missing`: list the missing lines

pytest-cov lists the lines no test ran

shell

```
~/todo $ uv add --dev pytest-cov
~/todo $ uv run pytest --cov=. --cov-report=term-missing
```

shell

```
Name          Stmts   Miss  Cover   Missing
-------------------------------------------
initials.py       4      0   100%
todo/app.py      31      6    81%   42-47
-------------------------------------------
TOTAL            35      6    83%
```

One row per file: its statements, how many never ran, and the share that did.

Lines 42 to 47 of `todo/app.py` **never ran.**

illustrative report

pytest-cov lists the lines no test ran

shell

```
~/todo $ uv add --dev pytest-cov
~/todo $ uv run pytest --cov=. --cov-report=term-missing
```

shell

```
Name          Stmts   Miss  Cover   Missing
-------------------------------------------
initials.py       4      0   100%
todo/app.py      31      6    81%   42-47
-------------------------------------------
TOTAL            35      6    83%
```

Start with the missing lines. Ask the agent to run the report and explain the gaps, and read the report yourself as well.

For lines that ran, go back to the assertions.

illustrative report

Keep, improve or remove each test

tests/test_initials.py

```python
def test_two_names():
    assert initials("Alan Turing") == "AT"

def test_two_names_2():
    assert initials("Ada Lovelace") == "AL"

def test_empty_name():
    assert initials("") == ""

def test_runs():
    initials("Grace Hopper")
```

Keep, improve or remove each test

tests/test_initials.py

```python
def test_two_names():
    assert initials("Alan Turing") == "AT"

def test_two_names_2():
    assert initials("Ada Lovelace") == "AL"

def test_empty_name():
    assert initials("") == ""

def test_runs():
    initials("Grace Hopper")
```

keep — first and last name

remove — the same case again

The second adds no distinct case, and it would fail for the same reasons.

Keep, improve or remove each test

tests/test_initials.py

```
 1 def test_two_names():
 2     assert initials("Alan Turing") == "AT"
 3
-def test_two_names_2():
-    assert initials("Ada Lovelace") == "AL"
 6
 7 def test_empty_name():
 8     assert initials("") == ""
 9
10 def test_runs():
11     initials("Grace Hopper")
```

keep — first and last name

remove — the same case again

keep — a different case

Keep, improve or remove each test

tests/test_initials.py

```
 1  def test_two_names():
 2      assert initials("Alan Turing") == "AT"
 3
 -  def test_two_names_2():
 -      assert initials("Ada Lovelace") == "AL"
 6
 7  def test_empty_name():
 8      assert initials("") == ""
 9
10  def test_runs():
11      initials("Grace Hopper")
```

keep — first and last name

remove — the same case again

keep — a different case

improve — add an expected value

It runs the code and checks nothing, so a wrong answer still passes.

Keep, improve or remove each test

tests/test_initials.py

```
 1 def test_two_names():
 2     assert initials("Alan Turing") == "AT"
 3
-def test_two_names_2():
-    assert initials("Ada Lovelace") == "AL"
 6
 7 def test_empty_name():
 8     assert initials("") == ""
 9
10 def test_runs():
-    initials("Grace Hopper")
+    assert initials("Grace Hopper") == "GH"
```

keep — first and last name

remove — the same case again

keep — a different case

improve — add an expected value

Keep, improve or remove each test

tests/test_initials.py

```
 1 def test_two_names():
 2     assert initials("Alan Turing") == "AT"
 3
-def test_two_names_2():
-    assert initials("Ada Lovelace") == "AL"
 6
 7 def test_empty_name():
 8     assert initials("") == ""
 9
10 def test_runs():
-    initials("Grace Hopper")
+    assert initials("Grace Hopper") == "GH"
```

keep — first and last name

remove — the same case again

keep — a different case

improve — add an expected value

Keep the tests whose expectations you understand and whose failures would tell you something.

A test suite is worth what it catches.

See you in the next video.