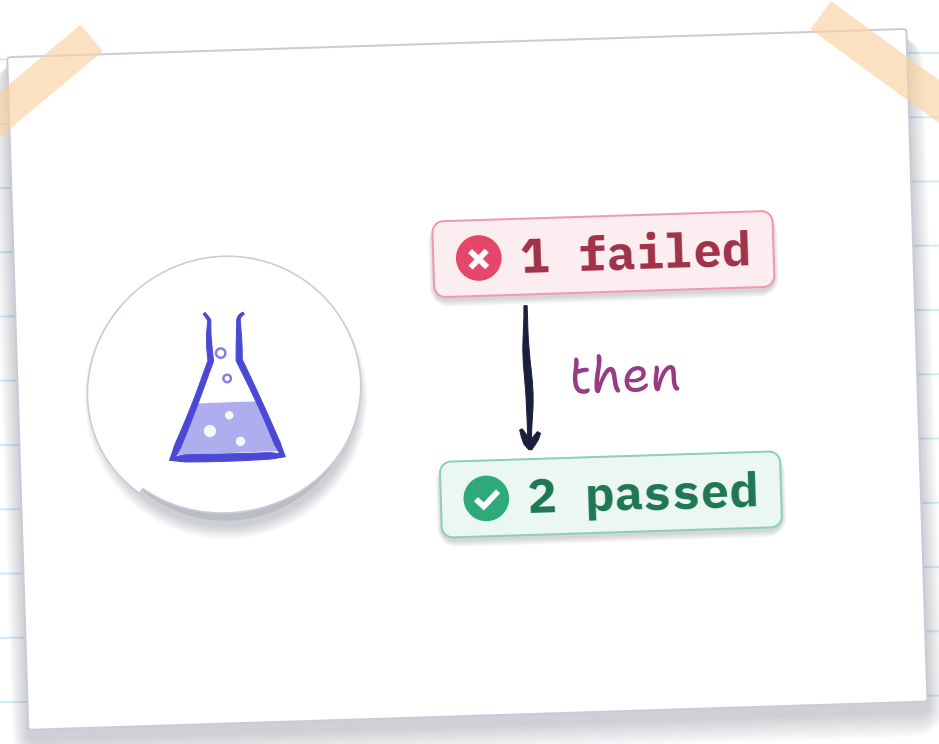
Prof. Aviv Zohar

Semester A, 2026–7

Testing with the agent

What a test checks, and how to write tests with an agent



A test compares a result with an expectation

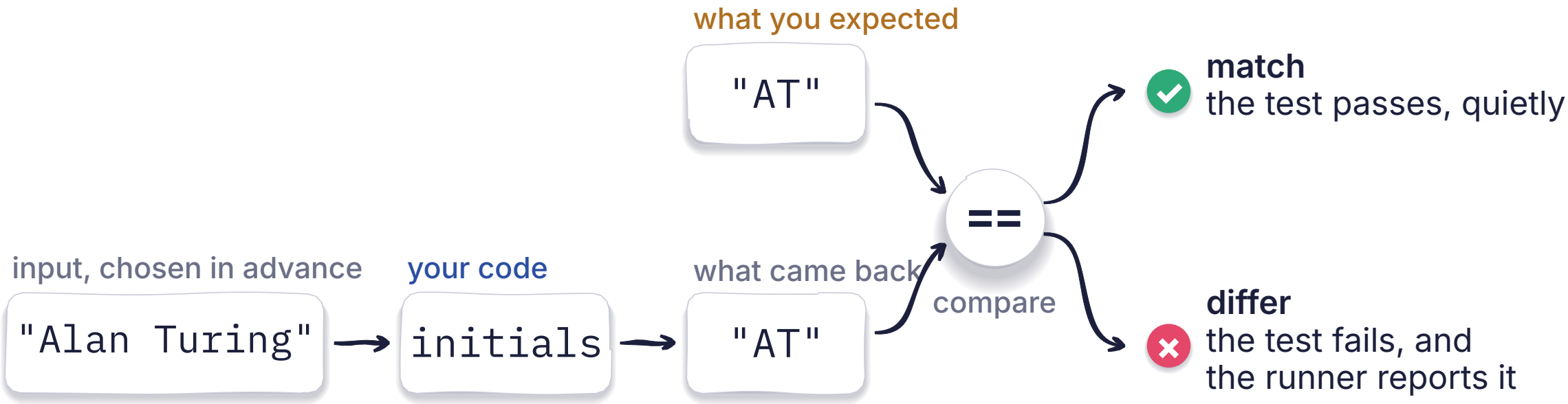


A test compares a result with an expectation

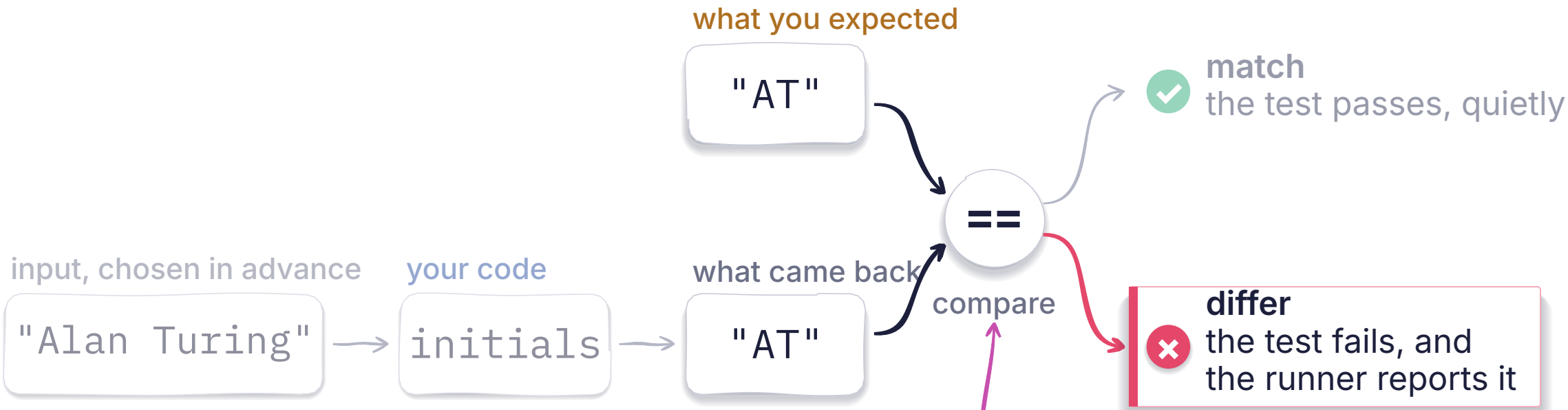




A test compares a result with an expectation



A test compares a result with an expectation



The point of a test is that it can fail.
When you first write one, watch it fail
for the reason you expect.

An assertion adds the missing comparison

Python, with pytest

test_initials.py

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

if initials returned `"ZZ"`



TypeScript, with vitest

initials.test.ts

```typescript
test("initials", () => {
  console.log(initials("Alan Turing"));
});
```



An assertion adds the missing comparison

Python, with pytest

test_initials.py

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

if initials returned `"ZZ"`

1 passed

Nothing compares the result.
It fails only on a crash.

TypeScript, with vitest

initials.test.ts

```typescript
test("initials", () => {
  console.log(initials("Alan Turing"));
});
```

1 passed

An assertion adds the missing comparison

Python, with pytest

test_initials.py

```
1 def test_initials():
-       print(initials("Alan Turing"))
+       got = initials("Alan Turing")
+       assert got == "AT"
```

if initials returned `"ZZ"`

✖ 1 failed

TypeScript, with vitest

initials.test.ts

```
1 test("initials", () => {
-   console.log(initials("Alan Turing"));
+   const got = initials("Alan Turing");
+   expect(got).toBe("AT");
4 });
```

✖ 1 failed

The assertion checks the value it got.

An assertion adds the missing comparison

Python, with pytest

test_initials.py

```
1 def test_initials():
-     print(initials("Alan Turing"))
+     got = initials("Alan Turing")
+     assert got == "AT"
```

TypeScript, with vitest

initials.test.ts

```
1 test("initials", () => {
-   console.log(initials("Alan Turing"));
+   const got = initials("Alan Turing");
+   expect(got).toBe("AT");
4 });
```

if initials returned `"ZZ"`

1 failed

print shows you the result while you explore or debug.

assert turns what you expect into a check the runner makes for you.

One command runs the tests

```shell
~/initials $ pytest
======= 2 passed in 0.01s =======

~/initials $ npx vitest run
 Tests  2 passed (2)
```

Python

TypeScript

Each runner finds the tests that the project's configuration selects, and runs them.

One command runs the tests

shell

```
~/initials $ pytest
======= 2 passed in 0.01s =======

~/initials $ npx vitest run
 Tests  2 passed (2)

~/initials $ pytest && git push
```

exit 0 — every test passed: `git push` runs

exit 1 — something failed: nothing is pushed

An agent reads the exit status to decide whether to go on, then reads the report to find out what needs fixing.

One command runs the tests

shell

```
~/initials $ pytest
======= 2 passed in 0.01s =======

~/initials $ npx vitest run
 Tests  2 passed (2)

~/initials $ pytest && git push

~/initials $ pytest -x
~/initials $ npx vitest run --bail 1
```

stop at the first failure

The report says what happened

shell

```
~/initials $ pytest
collected 2 items
test_initials.py .F

>       assert initials("ada lovelace") == "AL"
E       AssertionError: assert 'al' == 'AL'

FAILED test_initials.py::test_lowercase_name
====== 1 failed, 1 passed in 0.05s ======
```

ⓐ ⓑ ⓒ ⓓ

lower case came back;
upper case was expected

ⓐ the tests the runner selected: two

ⓑ one mark per test: . passed, F failed

ⓒ the assertion that did not hold, with both values

ⓓ the failing test, and the count

vitest's report has the same parts

The report says what happened

shell

```
~/initials $ pytest
collected 2 items
test_initials.py .F

>       assert initials("ada lovelace") == "AL"
E       AssertionError: assert 'al' == 'AL'

FAILED test_initials.py::test_lowercase_name
====== 1 failed, 1 passed in 0.05s ======
```

shell

```
~/initials $ pytest
collected 0 items / 1 error
ERROR collecting test_initials.py
E   ModuleNotFoundError: No module named 'names'
========== 1 error in 0.23s ==========
```

there is no module names, so the tests could not even load

The report says what happened

shell

```
~/initials $ pytest
collected 2 items
test_initials.py .F

>       assert initials("ada lovelace") == "AL"
E       AssertionError: assert 'al' == 'AL'

FAILED test_initials.py::test_lowercase_name
====== 1 failed, 1 passed in 0.05s ======
```

a failure

a verdict on the code under test

shell

```
~/initials $ pytest
collected 0 items / 1 error
ERROR collecting test_initials.py
E   ModuleNotFoundError: No module named 'names'
========== 1 error in 0.23s ==========
```

an error

a setup problem: no test ran,
so it says nothing about `initials`

Fix the setup first.

A bug becomes a test before it becomes a fix

initials.py

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

→ ['Alan', '', 'Turing']
an empty word
and ''[0] has no
first letter to take

```
initials("Alan  Turing")
```

→ IndexError

A bug becomes a test before it becomes a fix

initials.py

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

The agent can edit the code, rerun the test, and repeat until it passes.

`initials("Alan  Turing")` → **IndexError**

test_initials.py

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

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

FAILED · IndexError

the same crash:
the test reproduces it

A bug becomes a test before it becomes a fix

initials.py

```
1  def initials(name):
-      words = name.split(" ")
+      words = name.split()
3      return "".join(w[0] for w in words)
```

split on any run of whitespace

`initials("Alan  Turing")` → ✗ IndexError

test_initials.py

```
1  def test_initials():
2      assert initials("Alan Turing") == "AT"
3
4  def test_double_space():
5      assert initials("Alan  Turing") == "AT"
```

✓ 2 passed

and the original test still passes

A bug becomes a test before it becomes a fix

initials.py

```
1  def initials(name):
-      words = name.split(" ")
+      words = name.split()
3      return "".join(w[0] for w in words)
```

`initials("Alan  Turing")` → ⊗ **IndexError**

test_initials.py

```
1  def test_initials():
2      assert initials("Alan Turing") == "AT"
3
4  def test_double_space():
5      assert initials("Alan  Turing") == "AT"
```

regression test

Keep it in the suite, so later runs can detect the same bug returning.

Ask for the behaviour you care about

you

write tests for this

It leaves the agent to decide the scope, and what counts as correct.

Ask for the behaviour you care about

you

write tests for this

agent

Done: 40 new tests.

40 tests to read



What do they cover?
Which ones matter?

Ask for the behaviour you care about

you

write tests for this

agent

Done: 40 new tests.

40 tests to read



you

ⓐ Initials are uppercase, one letter per word, and extra spaces are ignored. ⓑ Start with the double-space case: ⓒ write the smallest failing test first, and stop there.

ⓐ say what must stay true
ⓑ name the cases you care about
ⓒ ask for the smallest failing case first

Ask for the behaviour you care about

you

write tests for this

agent

Done: 40 new tests.

40 tests to read



you

Initials are uppercase, one letter per word, and extra spaces are ignored. Start with the double-space case: write the smallest failing test first, and stop there.

`test_initials.py` written by the agent

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

You decide which properties matter. The agent can write the test; you review the expected answer.

Test-driven development: red, green, refactor

red
watch it fail

green
just enough to pass

refactor
keep the behaviour

one small behaviour per cycle

you

Write the test first. Show me the failing run, and explain why it fails.

```
FAILED test_double_space
```

It should fail because the behaviour is missing, not because of a setup error.

TDD takes discipline. With an agent, you can ask for each step and check that it happened.

Test-driven development: red, green, refactor



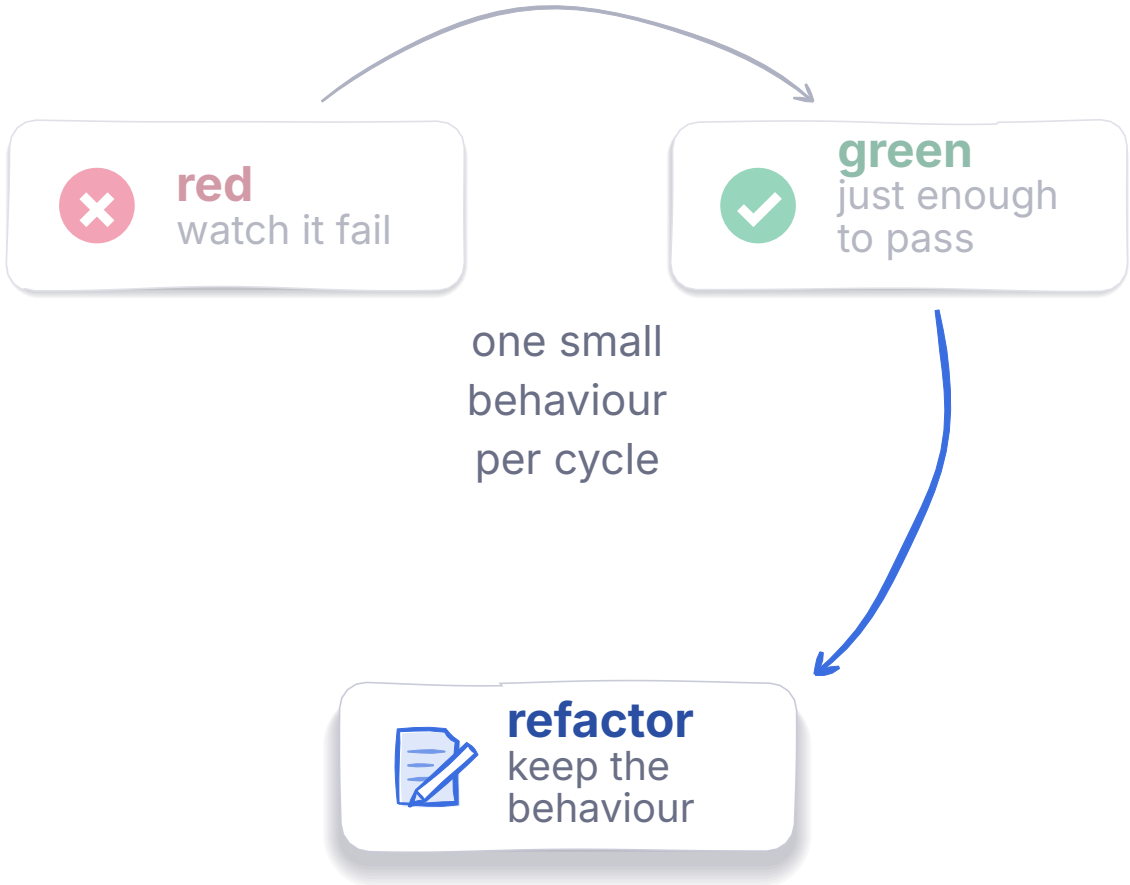
you

Write the least code that makes it pass.
Show me the passing run and the diff.

2 passed `split(" ")` → `split()`

just enough to turn it green,
not the design you imagine for later

Test-driven development: red, green, refactor



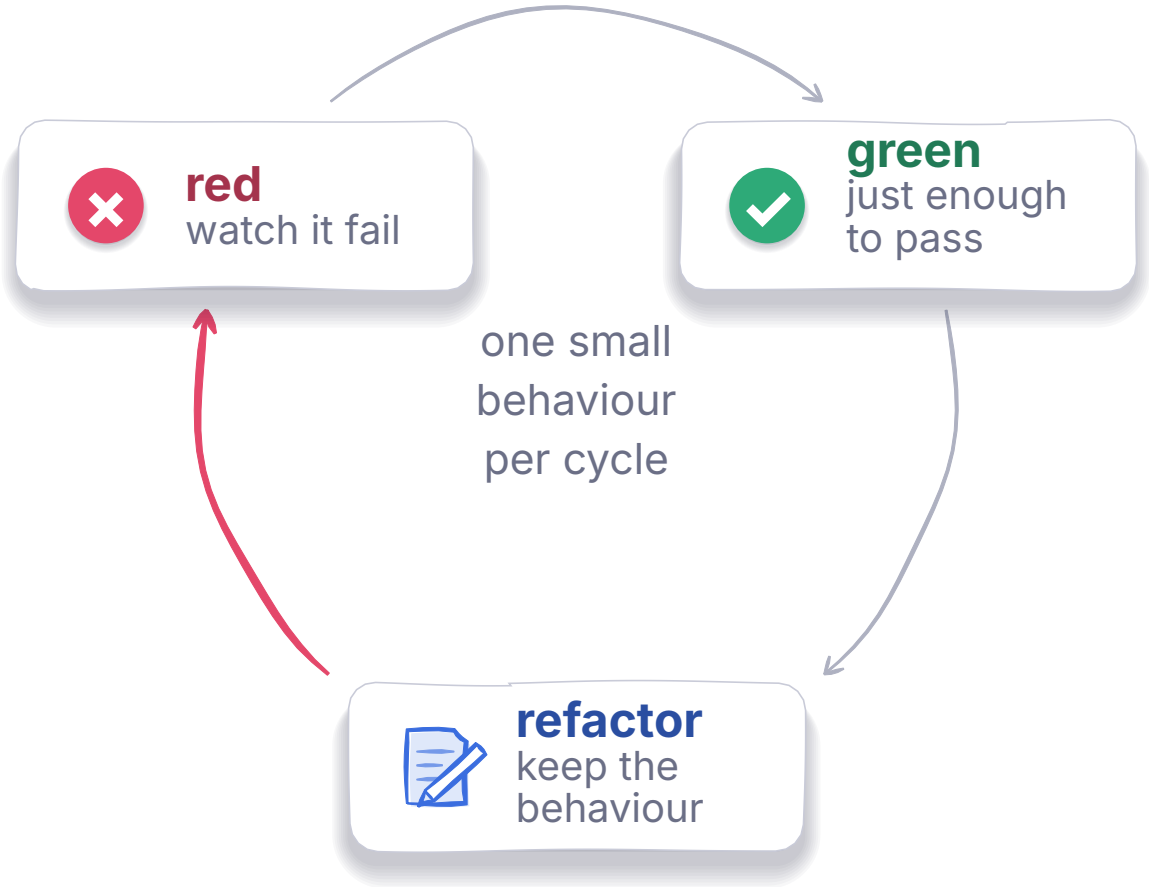
you

Clean up the structure, and keep every test passing.

2 passed

The tests protect the cases they cover while you move things around.

Test-driven development: red, green, refactor



next: a name typed in lower case

```
initials("ada lovelace") == "AL"
```

Each change starts with a failing test. Still look for the inputs you have not tested.

A test you have never seen fail is a test you have not tested.

Who writes which half

the code, written first

```
initials("ada lovelace") → "al"
```

reads it



the test, written after

```
assert initials("ada lovelace") ==
```

Who writes which half

the code, written first

```
initials("ada lovelace") → "al"
```

reads it

the test, written after

```
assert initials("ada lovelace") == "al"
```

1 passed

Code and test agree: they share the same misunderstanding.

The test passes, and the bug remains.

Who writes which half

the code, written first

```
initials("ada lovelace") → "al"
```

reads it

the test, written after

```
assert initials("ada lovelace") == "al"
```

you
the requirement
initials are uppercase

you
the test, first
expects `"AL"`: you write it or approve it

agent
the code
only what makes the test pass

you
the diff
you read it

You keep the part that decides what correct means;
the agent does the part it is fast at.

Who writes which half

the code, written first

```
initials("ada lovelace") → "al"
```

reads it

the test, written after

```
assert initials("ada lovelace") == "al"
```

if the request was misread, both can share one mistake

you
the requirement
initials are uppercase

you
the test, first
expects "AL": you write it or approve it

agent
the code
only what makes the test pass

you
the diff
you read it

Review the expected answer against the requirement.

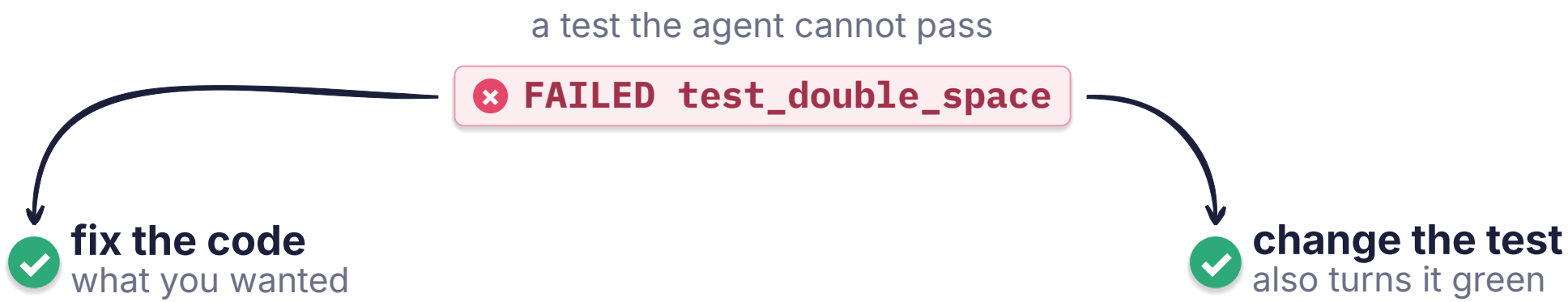
Reviewed, the test is a grader for new work, ready before the code exists.

A passing run can hide a weakened test

a test the agent cannot pass

```
FAILED test_double_space
```

A passing run can hide a weakened test



A passing run can hide a weakened test

a test the agent cannot pass

`FAILED test_double_space`

fix the code
what you wanted

change the test
also turns it green

initials.py — special-cased

```
    def initials(name):
+       if name == "Alan  Turing":
+           return "AT"
        words = name.split(" ")
```

test_initials.py — weakened

```
    def test_double_space():
-       assert initials("Alan  Turing") == "AT"
+       assert initials("Alan Turing") == "AT"
```

The test is untouched; the code returns `"AT"` for that one input.

Models have been observed removing assertions, hardcoding results, or special-casing the test input.

A passing run can hide a weakened test

a test the agent cannot pass

`FAILED test_double_space`

fix the code
what you wanted

change the test
also turns it green

initials.py — special-cased

```
    def initials(name):
+       if name == "Alan  Turing":
+           return "AT"
        words = name.split(" ")
```

test_initials.py — weakened

```
    def test_double_space():
-       assert initials("Alan  Turing") == "AT"
+       assert initials("Alan Turing") == "AT"
```

When the agent reports that the tests pass, **read the test diff** as well as the code diff, and look for special cases in the code.

A changed expectation needs a changed requirement.

Unit, integration, end to end



unit

one function, called directly in memory:
a millisecond or less, and a failure points to one place

Unit, integration, end to end



unit — one function, called directly in memory: a millisecond or less, and a failure points to one place

integration — your code and a real database, running and set up first. A failure has more than one suspect.

Unit, integration, end to end



unit — one function, called directly in memory: a millisecond or less, and a failure points to one place

integration — your code and a real database, running and set up first. A failure has more than one suspect.

end to end — the whole application, through its interface: server, data and browser ready first, and it can fail for reasons other than a defect in the behaviour

e.g. an animation still running when the test clicks

Unit, integration, end to end



unit — one function, called directly in memory: a millisecond or less, and a failure points to one place

integration — your code and a real database, running and set up first. A failure has more than one suspect.

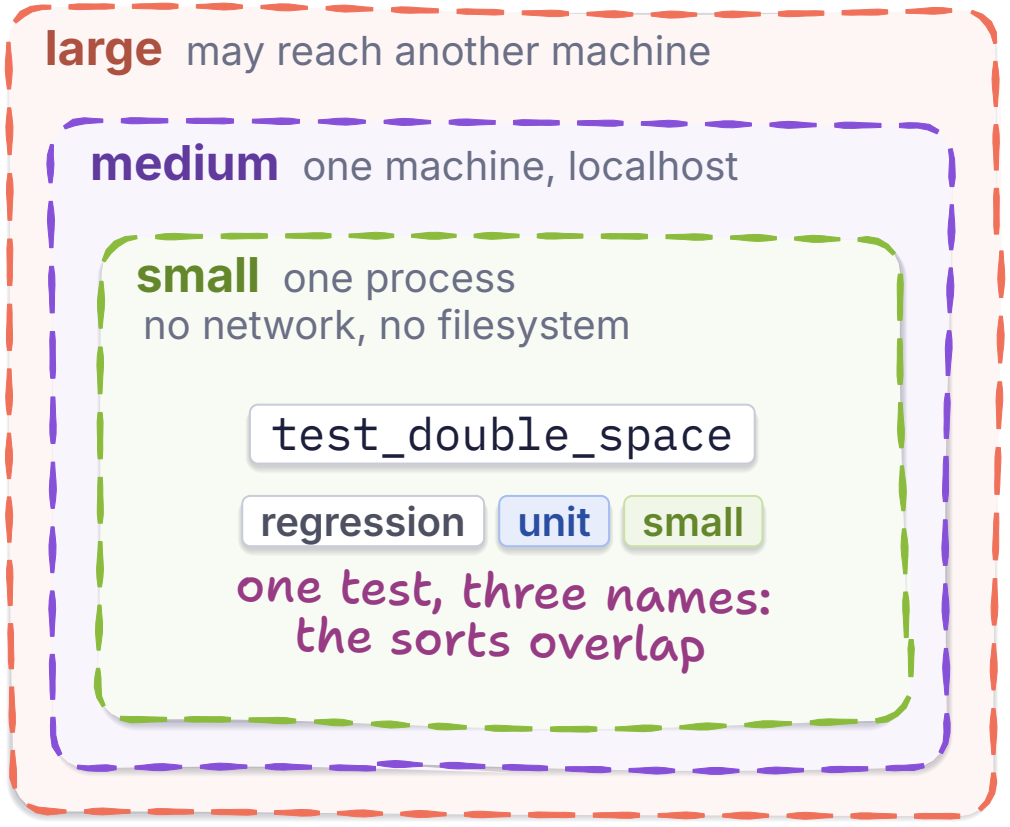
end to end — the whole application, through its interface: server, data and browser ready first, and it can fail for reasons other than a defect in the behaviour

Spend testing effort where a mistake is **likely or costly**, using the **smallest test** that can catch it.

many cases for the program's rules; broader tests for the connections

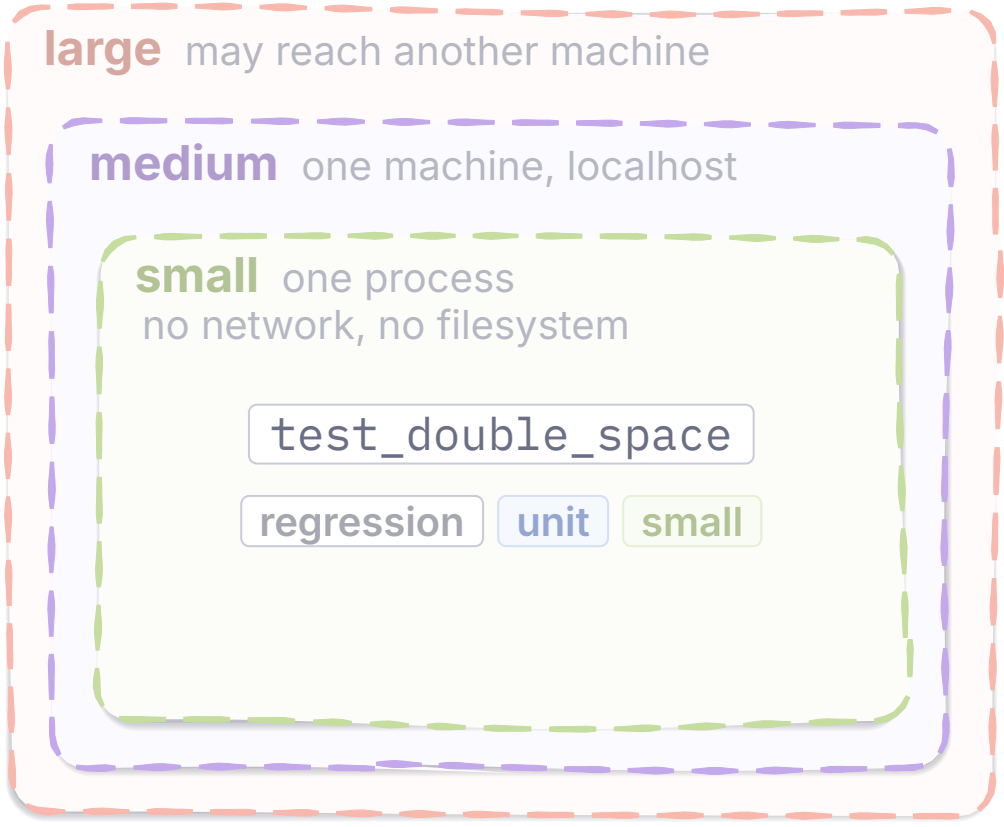
Two more ways to sort tests

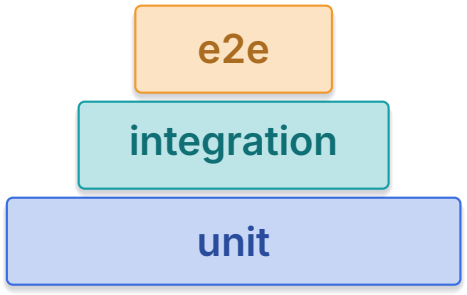
Google: by what a test may touch



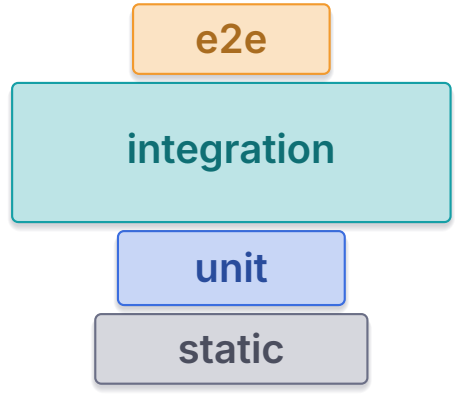
Two more ways to sort tests

Google: by what a test may touch





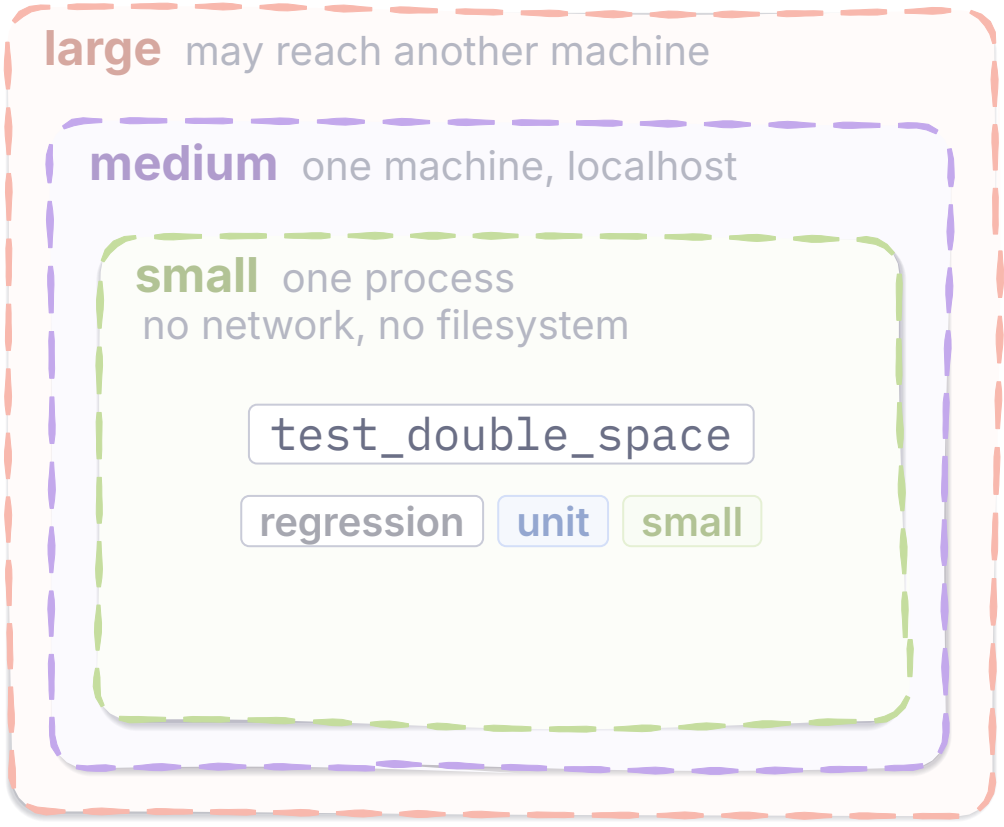
the testing pyramid
mostly unit tests

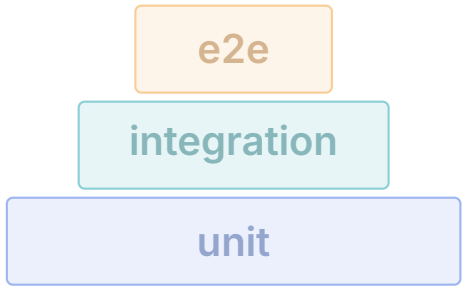


the testing trophy
static analysis at the base
(type checkers, linters),
then mostly integration tests

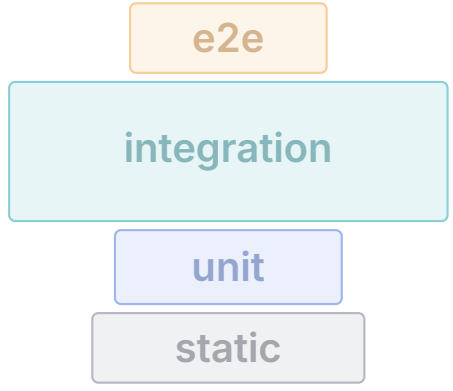
Two more ways to sort tests

Google: by what a test may touch





the testing pyramid
mostly unit tests



the testing trophy
static analysis at the base (type checkers, linters), then mostly integration tests

Neither shape fits every project. Weigh the risk in each part, the cost to write and run each kind, and how easily the tests break when the code changes.

There are many kinds of tests

property-based
a rule that must hold; inputs generated to break it

fuzzing
generated inputs, hunting for crashes

snapshot and golden
output compared with a saved, approved copy

There are many kinds of tests

property-based
a rule that must hold; inputs generated to break it

fuzzing
generated inputs, hunting for crashes

snapshot and golden
output compared with a saved, approved copy

visual regression
screenshots compared with saved pictures

performance, load, soak
speed under workloads, and over long periods

contract
services agree on their interface

There are many kinds of tests

property-based
a rule that must hold; inputs generated to break it

fuzzing
generated inputs, hunting for crashes

snapshot and golden
output compared with a saved, approved copy

visual regression
screenshots compared with saved pictures

performance, load, soak
speed under workloads, and over long periods

contract
services agree on their interface

security scanning
code, dependencies and exposed secrets

chaos engineering
infrastructure disrupted on purpose, to examine recovery

accessibility
keyboard access, and what a screen reader receives

There are many kinds of tests

property-based
a rule that must hold; inputs generated to break it

fuzzing @
generated inputs, hunting for crashes

snapshot and golden
output compared with a saved, approved copy

visual regression
screenshots compared with saved pictures

performance, load, soak
speed under workloads, and over long periods

contract
services agree on their interface

security scanning
code, dependencies and exposed secrets

chaos engineering
infrastructure disrupted on purpose, to examine recovery

accessibility
keyboard access, and what a screen reader receives

formal verification
a property proved for all cases of a model

manual exploratory
a person looking for problems nobody automated

Each kind answers a specific risk. @ A parser that accepts input from strangers is a good candidate for fuzzing.



1 failed

2 passed

Give the agent a specific behaviour to test, and check that its test catches the mistake.

Try it out. Happy testing!