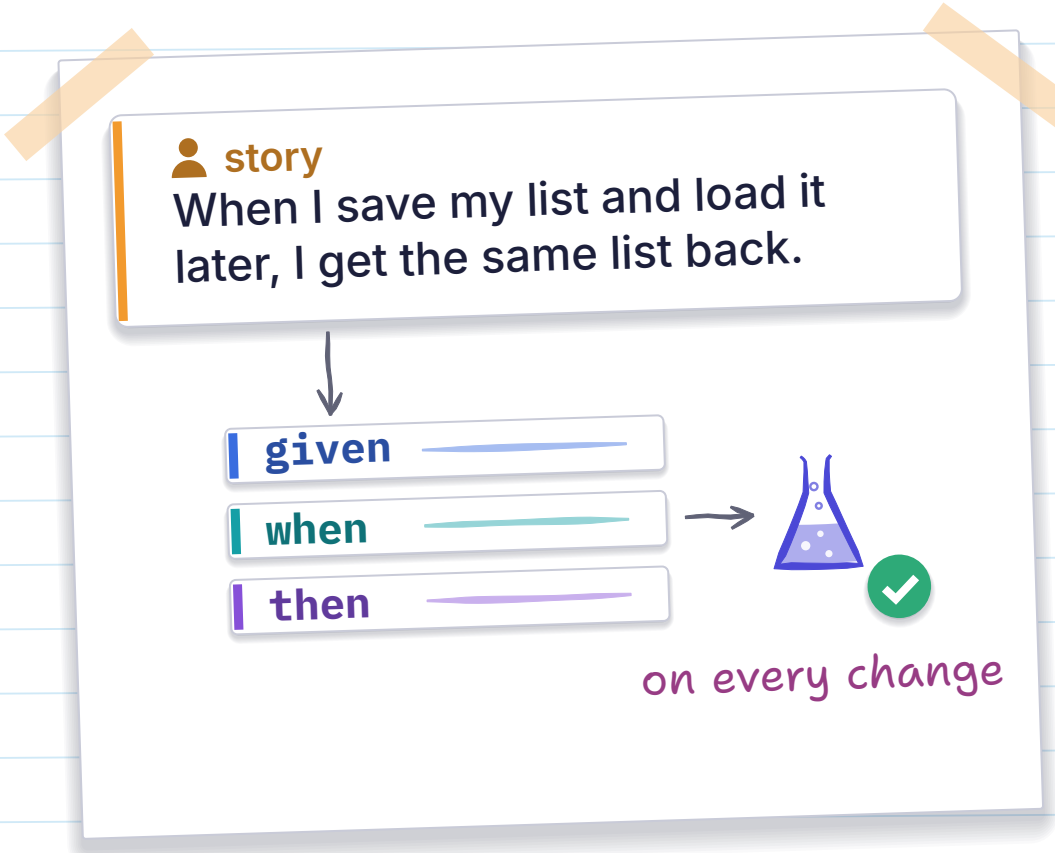
Prof. Aviv Zohar

Semester A, 2026–7

From Stories to Tests

Turning a requirement into a check that runs



A review by hand checks the spec once

SPEC.md — reviewed by hand

```
- New items go at the bottom, not yet done.
- Clicking a checkbox toggles the item done.
- Clearing completed items asks to confirm.
- The list looks clean and uncluttered.
```

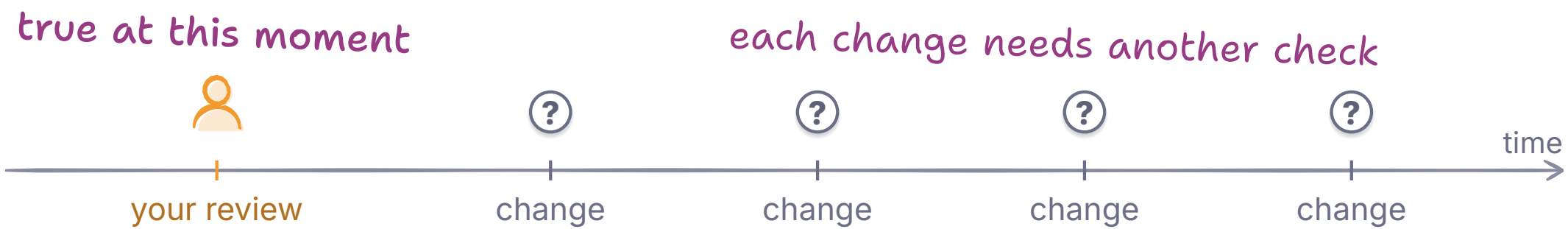
- met
- met
- needs a scope decision
- not met

A review by hand checks the spec once

SPEC.md — reviewed by hand

```
- New items go at the bottom, not yet done.
- Clicking a checkbox toggles the item done.
- Clearing completed items asks to confirm.
- The list looks clean and uncluttered.
```

- met
- met
- needs a scope decision
- not met

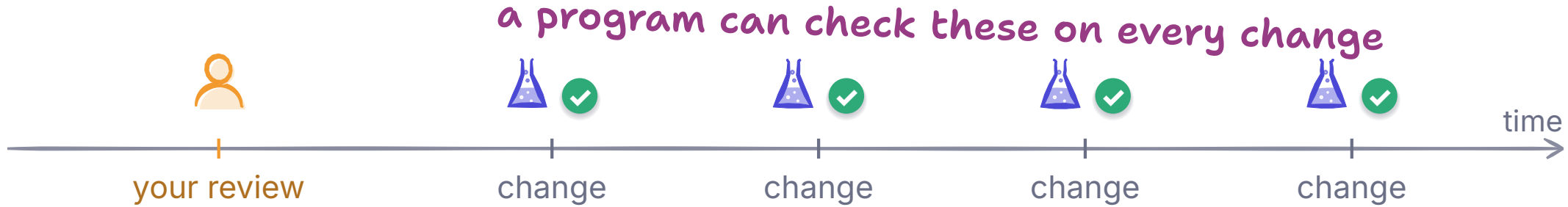


A review by hand checks the spec once

SPEC.md — reviewed by hand

```
- New items go at the bottom, not yet done.
- Clicking a checkbox toggles the item done.
- Clearing completed items asks to confirm.
- The list looks clean and uncluttered.
```

- met
- met
- needs a scope decision
- not met



A story says what the user does and what should happen

> story
>
> When I save my list and load it later,
> I get the same list back.

a requirement, written from the user's side

A story says what the user does and what should happen

story

When I save my list and load it later,
I get the same list back.

who what they do what should happen

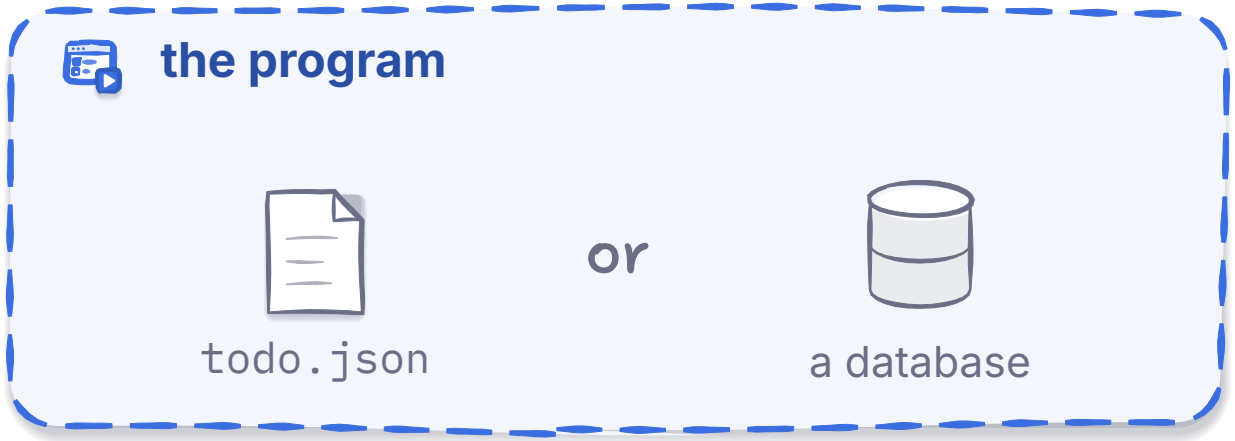
in your words, the way you would describe the app to a friend

A story says what the user does and what should happen

story

When I save my list and load it later,
I get the same list back.

who — what they do — what should happen

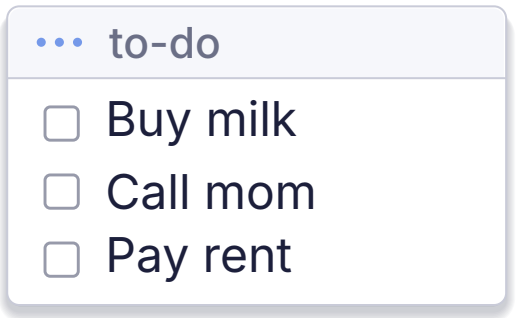


The story names no function, file format or framework.

The test chooses: a way into the program, and data to try.

A story becomes a test in three parts

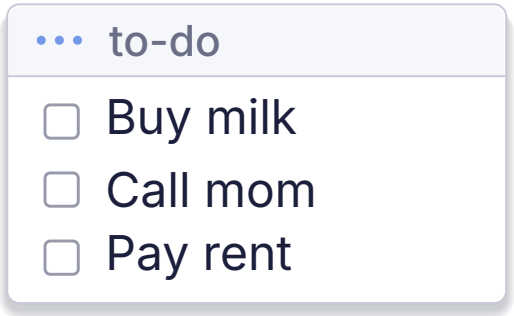
Set up the situation



a list with three open items

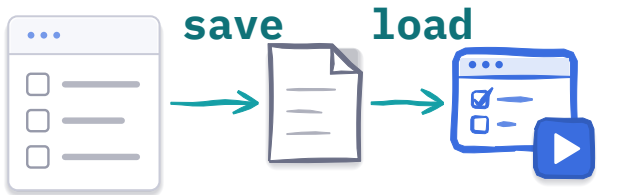
A story becomes a test in three parts

Set up the situation



a list with three open items

Perform the action



the list saved a fresh app

as after closing the app

A story becomes a test in three parts

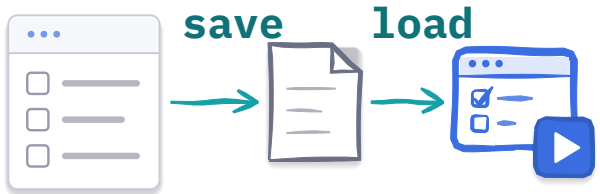
Set up the situation

to-do

- [ ] Buy milk
- [ ] Call mom
- [ ] Pay rent

a list with three open items

Perform the action



the list saved a fresh app

as after closing the app

Check the outcome

before		loaded
Buy milk	=	Buy milk
Call mom	=	Call mom
Pay rent	=	Pay rent

same items, same order

One behaviour per test, so a failure points at one thing to investigate.

A story becomes a test in three parts

given

Set up the situation

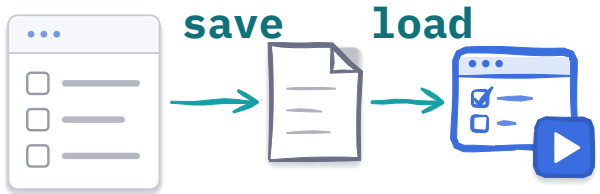
to-do

- [ ] Buy milk
- [ ] Call mom
- [ ] Pay rent

a list with three open items

when

Perform the action



as after closing the app

then

Check the outcome

before		loaded
☐ Buy milk	=	☐ Buy milk
☐ Call mom	=	☐ Call mom
☐ Pay rent	=	☐ Pay rent

same items, same order

Names from behaviour-driven development, used here in an ordinary test.

The same three parts, written in pytest

tests/test_todo.py

```
4 def test_saved_list_loads_back_the_same():
5     items = list_with(open=["Buy milk", "Call mom", "Pay rent"])
6     restored = load(save(items))
7     assert restored == items
```

given

when

then

== checks every item, every field, and their order

The same three parts, written in pytest

tests/test_todo.py

```
4 def test_saved_list_loads_back_the_same():
5     items = list_with(open=["Buy milk", "Call mom", "Pay rent"])
6     restored = load(save(items))
7     assert restored == items
```

given
when
then

story

When I save my list and load it later, I get the same list back.

`test_json_roundtrip`

names the code, not what the user gets back

The same three parts, written in pytest

tests/test_todo.py

```
4 def test_saved_list_loads_back_the_same():                    ⓐ
5     items = list_with(open=["Buy milk", "Call mom", "Pay rent"])
6     restored = load(save(items))
7     assert restored == items
```

given
when
then

shell

```
~/todo $ uv run pytest -q
⋮
E         Right contains one more item: Item(title='Pay rent', done=False)
⋮
FAILED tests/test_todo.py::test_saved_list_loads_back_the_same   ⓐ
1 failed in 0.20s
```

ⓑ Pay rent was lost on the way back: start with the code that reads the list.
ⓐ The report names the test, and with it the requirement.

Two features that pass alone can fail together

Complete an item — its test passes

- [x] ~~Buy milk~~
- [ ] Call mom

Its checkbox marks it done.
Active hides done items.

Save and load — its test passes

- [ ] Buy milk
- [ ] Call mom
- [ ] Pay rent

Loads back the same list.
its list: nothing done

to-do

All **Active** Done

- [ ] Call mom

Buy milk is done, so Active hides it.

Two features that pass alone can fail together

Complete an item

its test passes

- [x] ~~Buy milk~~
- [ ] Call mom

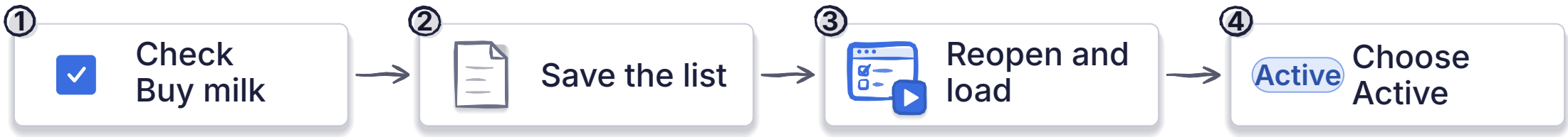
Its checkbox marks it done.
Active hides done items.

Save and load

its test passes

- [ ] Buy milk
- [ ] Call mom
- [ ] Pay rent

Loads back the same list.



Two features that pass alone can fail together

Complete an item — its test passes

- [x] ~~Buy milk~~
- [ ] Call mom

Its checkbox marks it done.
Active hides done items.

Save and load — its test passes

- [ ] Buy milk
- [ ] Call mom
- [ ] Pay rent

Loads back the same list.

① Check Buy milk → ② Save the list → ③ Reopen and load → ④ Active Choose Active

after loading

All **Active** Done

- [ ] Buy milk
- [ ] Call mom

Expected: Call mom alone.
Buy milk is back.

The file keeps the title, and nothing else.

todo.json — saved

```
[
  {"title": "Buy milk"},
  {"title": "Call mom"}
]
```

Two features that pass alone can fail together

Complete an item — its test passes

- [x] ~~Buy milk~~
- [ ] Call mom

Its checkbox marks it done. Active hides done items.

Save and load — its test passes

- [ ] Buy milk
- [ ] Call mom
- [ ] Pay rent

Loads back the same list.

① Check Buy milk → ② Save the list → ③ Reopen and load → ④ Active Choose Active

sets the done flag — leaves it out — comes back not done — reads it

after loading

All **Active** Done

- [ ] Buy milk
- [ ] Call mom

The done flag is state: save has to carry it too.

Neither test saved an item that was done.

todo.json saved

```
[
  {"title": "Buy milk"},
  {"title": "Call mom"}
]
```

A test for the combination

tests/test_todo.py

```
10 def test_completed_item_stays_hidden_after_save_and_load():
11     items = list_with(open=["Buy milk", "Call mom"])
12     items = complete(items, "Buy milk")
13     restored = load(save(items))
14     assert visible_items(restored, filter="active") == ["Call mom"]
```

given

when

then

```
assert ['Buy milk', 'Call mom'] == ['Call mom']
```

`visible_items` decides what the Active filter shows.

Several steps, one result. Buy milk comes back as not done.

A test for the combination

tests/test_todo.py

```
10 def test_completed_item_stays_hidden_after_save_and_load():
11     items = list_with(open=["Buy milk", "Call mom"])
12     items = complete(items, "Buy milk")
13     restored = load(save(items))
14     assert visible_items(restored, filter="active") == ["Call mom"]
```

given

when

then

2 passed

after the fix: save writes the done flag, and load reads it back

todo.json — after the fix

```
- {"title": "Buy milk"},
+ {"title": "Buy milk", "done": true},
```

Keep the test. It catches a done flag that goes missing again.

Outcomes get tests; judgments get review

SPEC.md

```
- Only unfinished items show under Active.
- The list looks clean and uncluttered.
- Clicking a checkbox marks the item done.
- Done items are easy on the eye.
- An empty item cannot be added.
- Adding an item feels quick.
```



A test

sets up a case and compares the result with an expectation.

Outcomes get tests; judgments get review

SPEC.md

```
- Only unfinished items show under Active.
- The list looks clean and uncluttered.
- Clicking a checkbox marks the item done.
- Done items are easy on the eye.
- An empty item cannot be added.
- Adding an item feels quick.
```

A target: a new item appears within 0.1 s. A test can check that number. Someone still judges whether it feels quick.

A test

sets up a case and compares the result with an expectation.

A review

someone uses the app and judges how it looks and feels.

Outcomes get tests; judgments get review

SPEC.md

```
- Only unfinished items show under Active.
- The list looks clean and uncluttered.
- Clicking a checkbox marks the item done.
- Done items are easy on the eye.
- An empty item cannot be added.
- Adding an item feels quick.
```



Both kinds stay in the spec.

Sorting them as you write plans the tests and the review.



A test

sets up a case and compares the result with an expectation.



A review

someone uses the app and judges how it looks and feels.

Recap



A behaviour clause becomes a named test

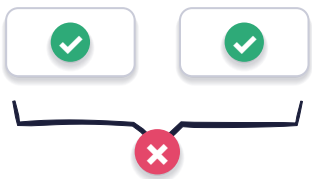
It runs after each change, a repeatable check of that requirement.
Review time goes to what still needs judgment.

Recap



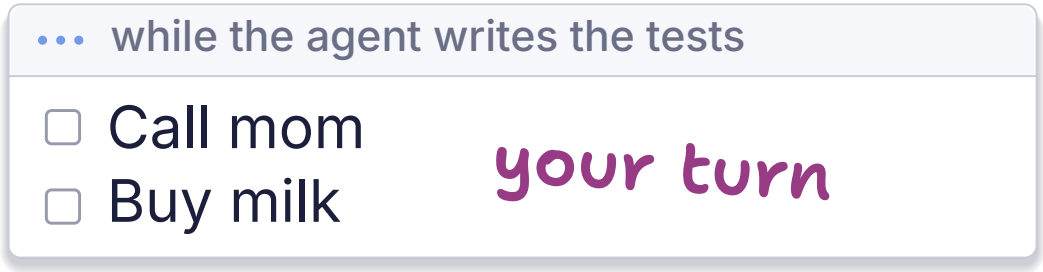
A behaviour clause becomes a named test

It runs after each change, a repeatable check of that requirement. Review time goes to what still needs judgment.



Test features together when they share state

Features can pass their own tests and still fail together. Test the combined behaviour, and keep that check for later changes.



See you in the next video.