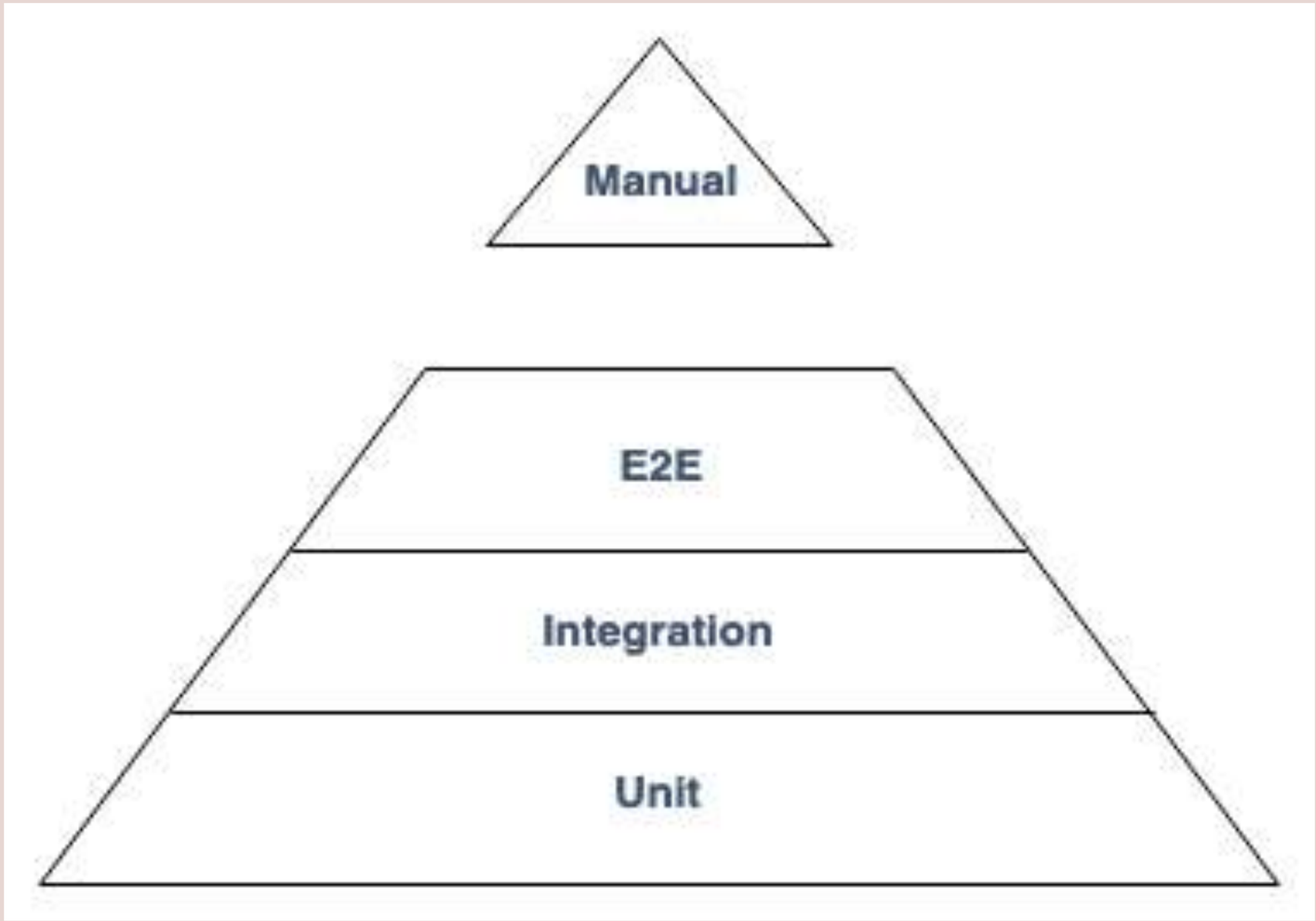
TESTING REACT COMPONENTS

WHY AUTOMATED TESTING

- manual testing is expensive
  - long feedback loop between developer/testers
- helps to find and prevent defects
- allow to change a systems behavior predictably
- find/prevent defects
- gain confidence about quality of the software

TESTING PYRAMID



TESTING PYRAMID

- Unit tests
  - lots of small and isolated tests which are fast to execute
- Integration tests
  - some integration tests which test external systems like databases
- E2E
  - few tests which test the whole system

UNIT TESTING AND TDD

- Test driven development (also known as TDD)
- Type of software development
- Introduced by Kent Beck
  - Author of Extreme Programming

WHY TDD

- early/fast feedback during development
- Driving the design of our application
  - Testing is a side-effect
- Possibility to refactor
  - Confidence that app is still working
- Break down large problems into small problems
  - Think about edge cases

TDD TO ME

“Helps me to break down bigger tasks into small steps I can keep in my head”

TDD

“TDD doesn't drive good design. TDD gives you immediate feedback about what is likely to be bad design. (Kent Beck)
I want to go home on Friday and don't think I broke something. (Kent Beck)”

WHAT IS TDD NOT

- Silver bullet for clean code
  - it eventually leads to better code
- Replacement for other testing strategies
  - TDD doesn't catch all bugs
  - Helps adding regression tests

THE BEST TDD CAN DO, IS ASSURE THAT THE CODE DOES WHAT THE DEVELOPER THINKS IT SHOULD DO. (JAMES GRENNING)

TDD INTRO IN 7:26 MINUTES

https://www.youtube.com/watch?v=WSes_PexXcA

ESSENTIAL VS. ACCIDENTAL COMPLICATION

- Essential complication
  - The problem is hard
  - eg. Tax return Software
  - nothing we can do about
- Accidental complication
  - We are not so good in our job
  - eg. future proofing code

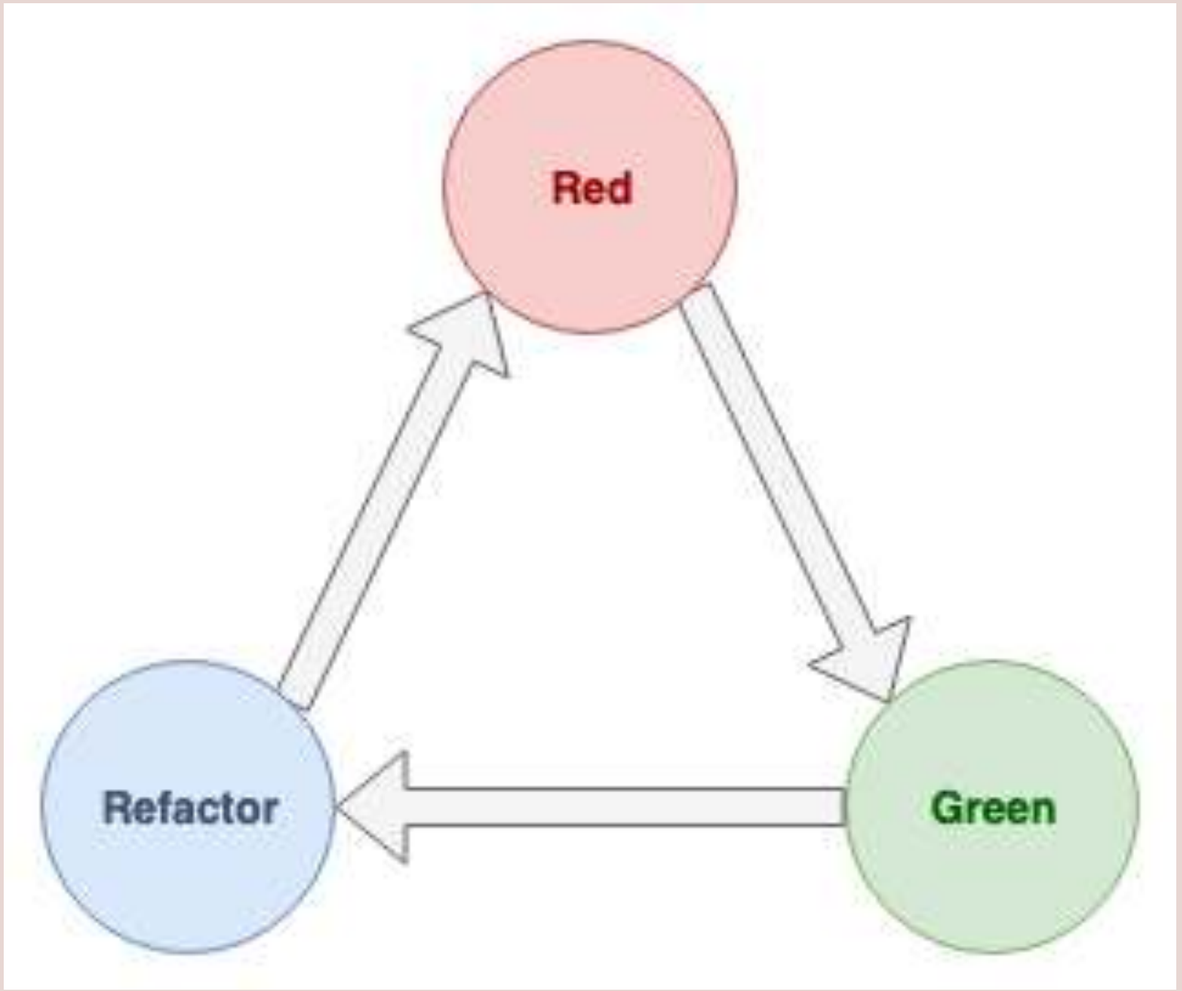
ACCIDENTAL COMPLICATION

- future proofing code
- cutting corners
  - to get stuff out of the door
  - we're not going to change this anyways
- drives up the cost/development time of a feature
  - mostly the feature isn't complex
  - the way the app is built drives the cost of a

AVOID ACCIDENTAL COMPLICATION

- Baby steps and TDD
- Refactor a little after every feature/green test
  - clean the kitchen
  - prevents big bang refactoring
  - which are hard to sell to business
- Without refactoring features will take longer

TDD CYCLE



TDD CYCLE

» Red: Write a test and watch it fail

» Green: Write just as much code to make the test pass

» Refactor: Clean up

RED

» Think about the test description

» Descriptions should reflect the behaviour of the program

```
it('when product A given, price is 3$', () => {
  expect(calculatePrice('productA')).toEqual('3$')
})
```

GREEN

» Write just enough code to make the test pass

  » if there is only 1 product just return 3$

```
function caluculatePrice () {
  return '3$'
};

it('when product A given, price is 3$', () => {
  expect(calculatePrice('productA')).toEqual('3$')
})
```

REFACTOR

» Change the code without changing any of the behaviour

» "Clean the kitchen"

```
const caluculatePrice = () => '3$'

it('when product A given, price is 3$', () => {
  expect(calculatePrice('productA')).toEqual('3$')
})
```

RED (REPEAT)

» start with the next test-case

```
const caluculatePrice = () => '3$'

it('when product A given, price is 3$', () => {
  expect(calculatePrice('productA')).toEqual('3$')
})

it('when product b given, price is 10$', () => {
  expect(calculatePrice('productB')).toEqual('10$')
})
```

GREEN

» start with the next test-case

```
const caluculatePrice = (product) => {
  if (product === 'productA') { return '3$' }
  if (product === 'productB') { return '10$' }
}

it('when product A given, price is 3$', () => {
  expect(calculatePrice('productA')).toEqual('3$')
})

it('when product B given, price is 10$', () => {
  expect(calculatePrice('productB')).toEqual('10$')
})
```

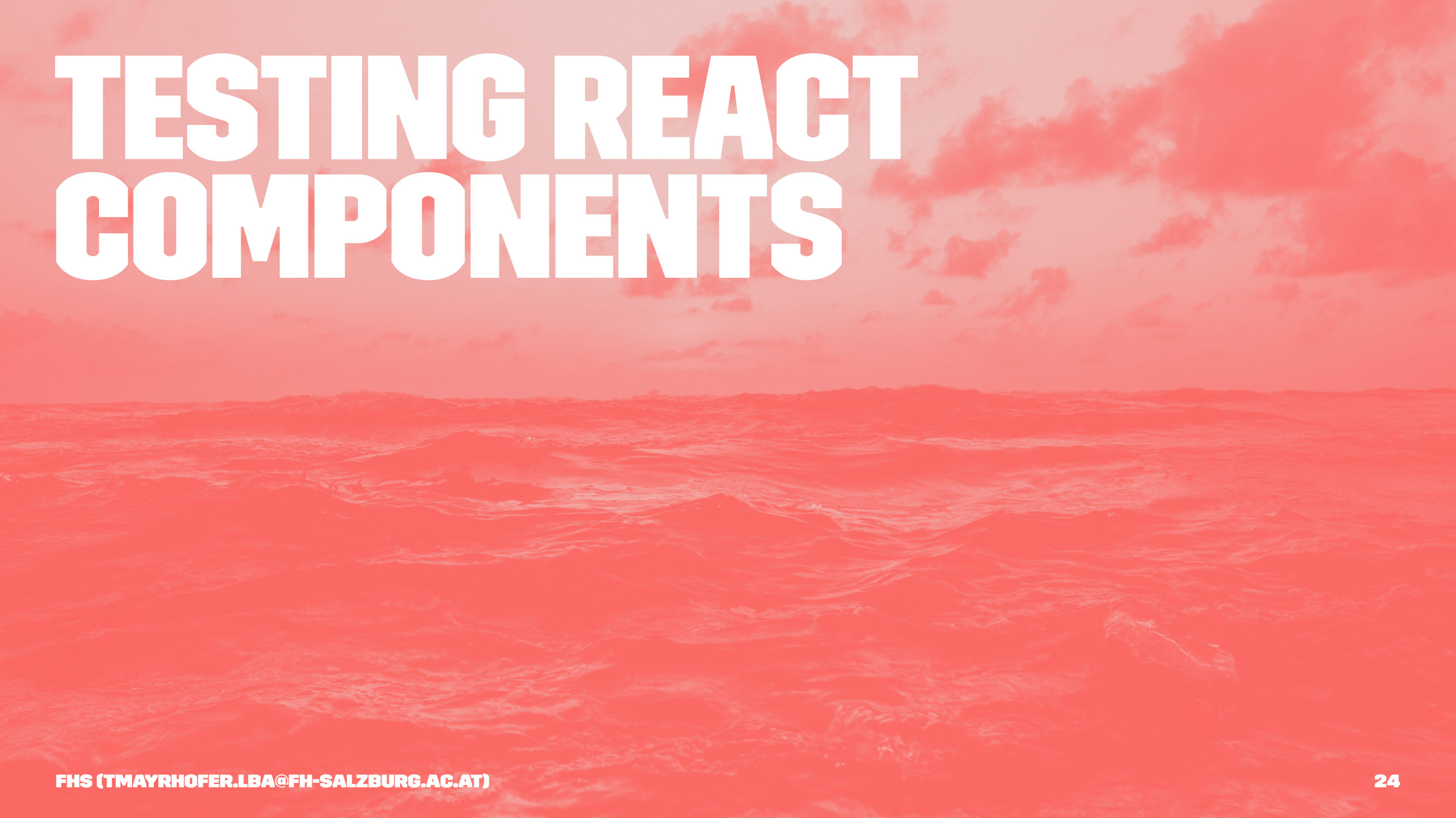
RED

```
const caluculatePrice = (product) => {
  if (product === 'productA') { return '3$' }
  if (product === 'productB') { return '10$' }
}

it('when product A given, price is 3$', () => {
  expect(calculatePrice('productA')).toEqual('3$')
})

it('when product B given, price is 10$', () => {
  expect(calculatePrice('productB')).toEqual('10$')
})

it('when unknown product is given, throws UnknownProductError', () => {
  expect(() => calculatePrice('productB')).toThrow(UnknownProductError)
})
```

GREEN

```
class UnknownProductError extends Error {}

const caluculatePrice = (product) => {
  if (product === 'productA') { return '3$' }
  if (product === 'productB') { return '10$' }
  throw new UnknownProductError();
}

it('when product A given, price is 3$', () => {
  expect(calculatePrice('productA')).toEqual('3$')
})

it('when product B given, price is 10$', () => {
  expect(calculatePrice('productB')).toEqual('10$')
})

it('when unknown product is given, throws UnknownProductError', () => {
  expect(() => calculatePrice('productB')).toThrow(UnknownProductError)
})
```

TESTING REACT COMPONENTS



TESTING REACT COMPONENTS
TOOLS

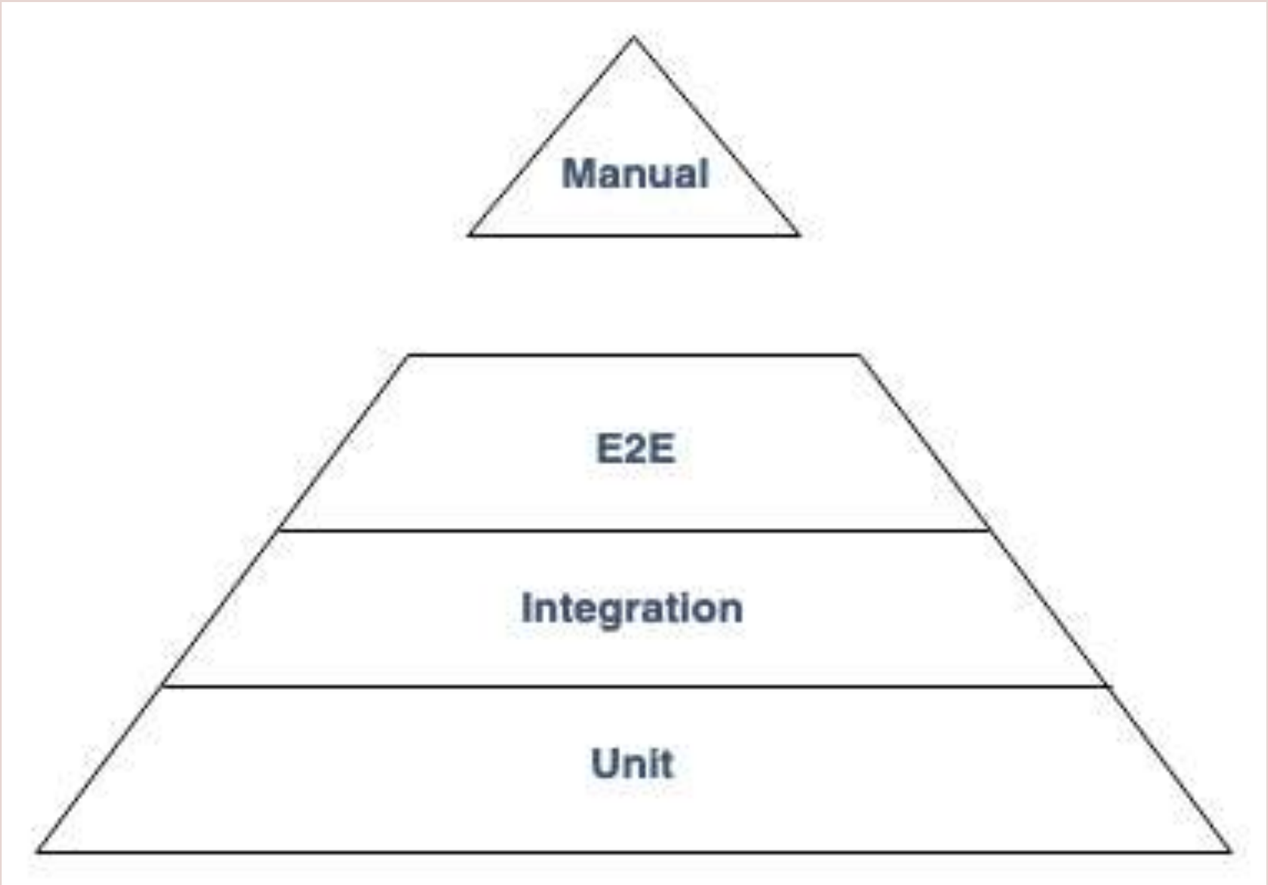
» Cypress (e2e testing "Software Quality Assurance")

» React Test Utils (Facebook, low level)

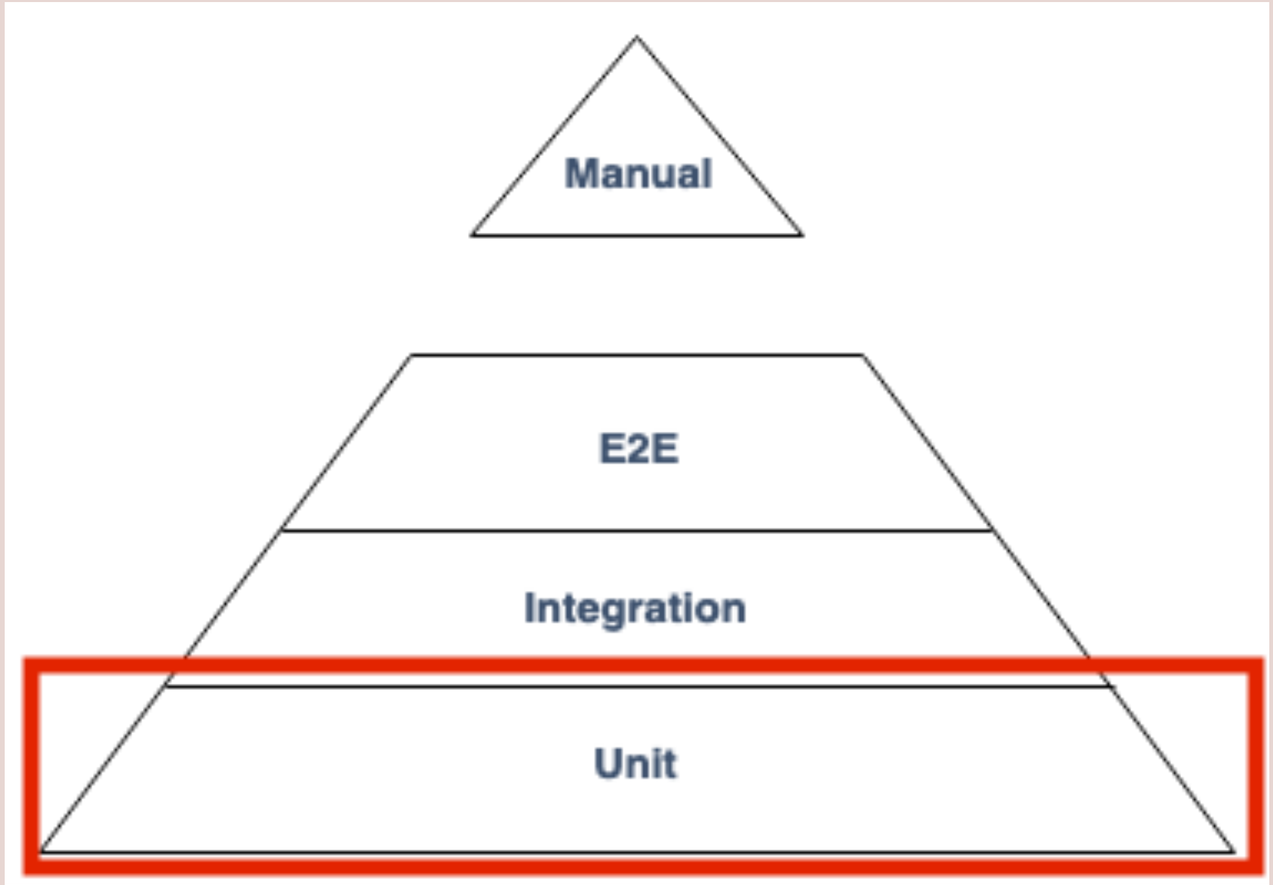
» Enzyme (AirBnB, more high level)

» React Testing Library (best of both worlds)

TESTING REACT COMPONENTS
TESTING PYRAMID



TESTING REACT COMPONENTS
TESTING PYRAMID



TESTING COMPONENTS
VERIFY THAT SOME TEXT IS RENDERED

```
// button.spec.js
import React from 'react'
import { render, cleanup, queryByText } from '@testing-library/react'
import { Button } from './Button'

afterEach(cleanup)
describe('Button', () => {
  it('renders given text', () => {
    // Arrange
    const givenText = 'A Button'

    // Act
    const { container } = render(<Button>{givenText}</Button>)

    // Assert
    expect(queryByText(container, givenText)).toBeTruthy()
  })
})
```

TESTING COMPONENTS
VERIFY THAT SOME TEXT IS RENDERED

» Run the test and watch it fail

» The test will guide you towards the implementation

TESTING COMPONENTS
VERIFY THAT SOME TEXT IS RENDERED

```
// button.js
import React from 'react'

const Button = ({ children }) => {
  return <>{children}</>
}
```

TESTING COMPONENTS
VERIFY THAT AN HTML TAG IS RENDERED

```
// button.spec.js
import React from 'react'
import { render, cleanup, queryByText } from '@testing-library/react'
import { Button } from './Button'

afterEach(cleanup)
describe('Button', () => {
  // ... other tests

  it('renders a button html tag', () => {
    const givenText = 'A Button'
    const { container } = render(<Button>{givenText}</Button>)

    expect(container.querySelectorAll('button'))
// 1)                 ^^^^^^^^^^^^^^^^^
      .toHaveProperty("length", 1)
// 2)  ^^^^^^^^^^^^^

// 1) get all button HTML tags from the rendered container
// 2) verify that there is exactly one button HTML tag
  })
})
```

TESTING COMPONENTS
VERIFY THAT AN HTML TAG IS RENDERED

```
import React from 'react'

export const Button = ({ children }) => (
  <button> { /* render a Button instead of a fragment */ }
    {children}
  </button>
)
```

TESTING USER INTERACTIONS

TESTING USER INTERACTIONS

» How can we test user interactions?

  » eg. when the button was clicked?

```
import React from 'react'

const Button = ({ onClick, children }) => {
  //                ^^^^^^^
  // How can we verify that the onClick handler is
  // called when the button is clicked?
  return <button>{children}</button>
}
```

TESTING USER INTERACTIONS
SPY OBJECTS

» Spy objects are stubs that also record the way they were called

» Useful when:

  » A hard to verify side effect is triggered (eg. E-Mail sending)

```
it('sends an email on sign up', () => {
  const sendEmail = jest.fn()
  const signUp = signUp({ sendEmail }, username, password)
  expect(sendEmail).toHaveBeenLastCalledWith({ username, password })
})
```

TESTING USER INTERACTIONS
SPY OBJECTS

» Spy objects can be used to verify user interactions in react components

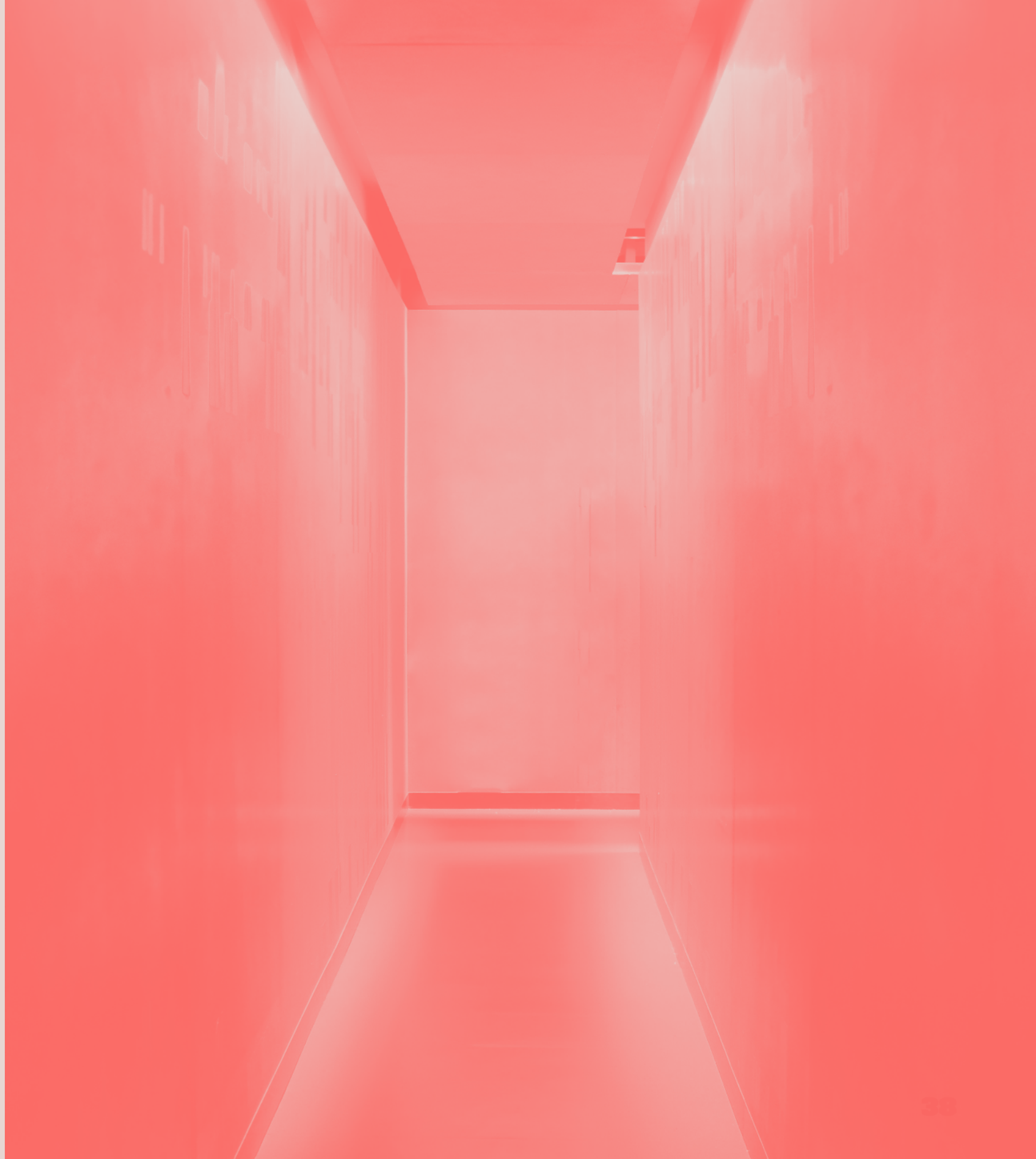
```
// button.spec.js

  it('when button was clicked, calls given onClick handler', () => {
    const onClick = jest.fn() // create function spy
    const { container } = render(<Button onClick={onClick}>My Button</Button>)
    fireEvent.click(container.querySelector('button'))
// 1)                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// 2)         ^^^^^
// 1) select the DOM element to be clicked
// 2) fire the click event

    expect(onClick).toHaveBeenCalled()
//  verify that the onClick spy has been called at least once
  })
```

TESTING USER INTERACTIONS
SPY OBJECTS

```
import React from 'react'

const Button = ({ onClick, children }) => (
  <button onClick={onClick}>
{/* 1)                 ^^^^^^^   */}
    {children}
  </button>
)
// 1) set click handler on the button DOM element
```

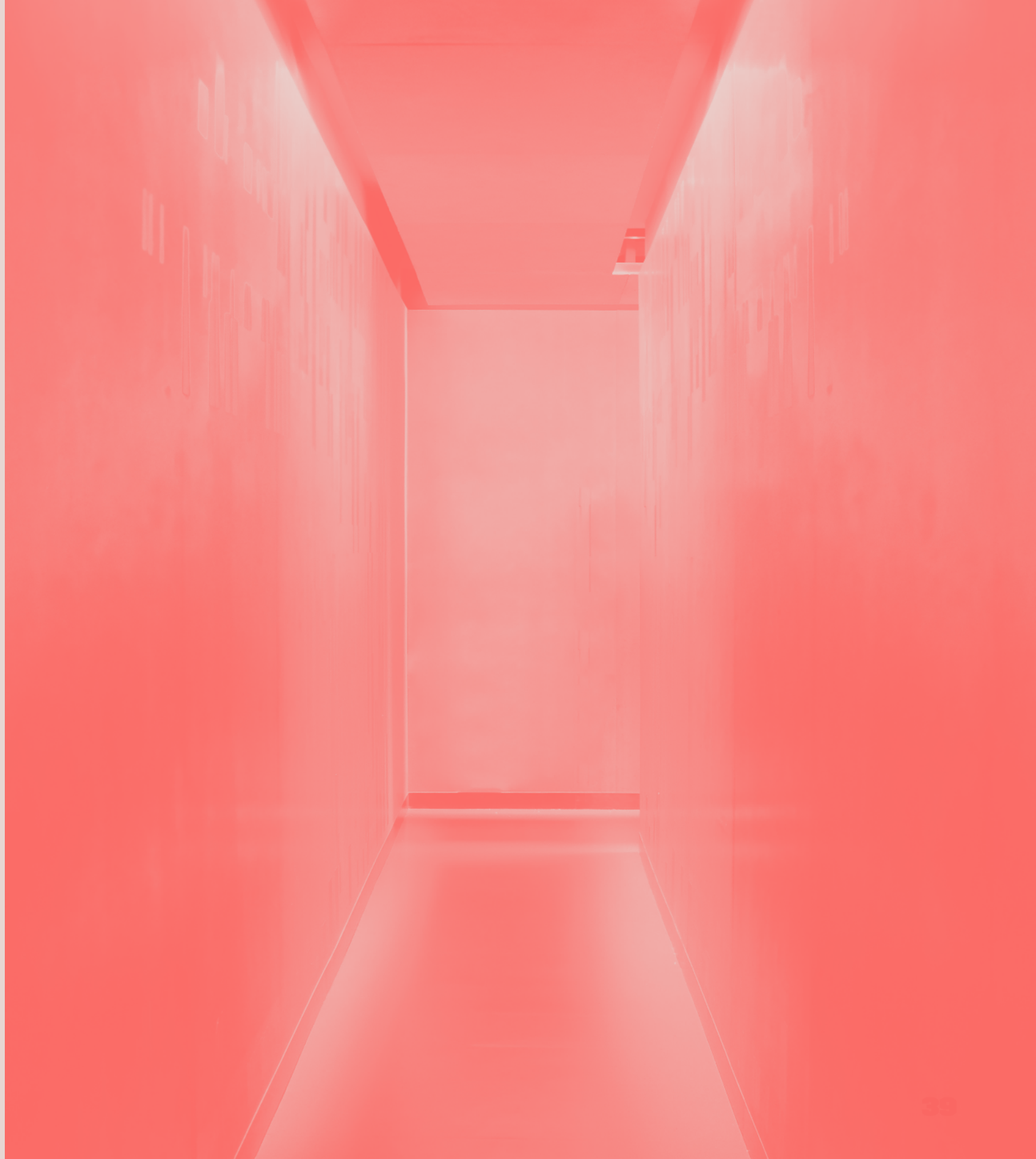
EXERCISE 1

- You're building an issue tracking system
- Build a component which displays the names of assignees
  - eg. `<Assignees assignees={['Mike', 'Sepp', 'David']} />`
  - build a simple ul



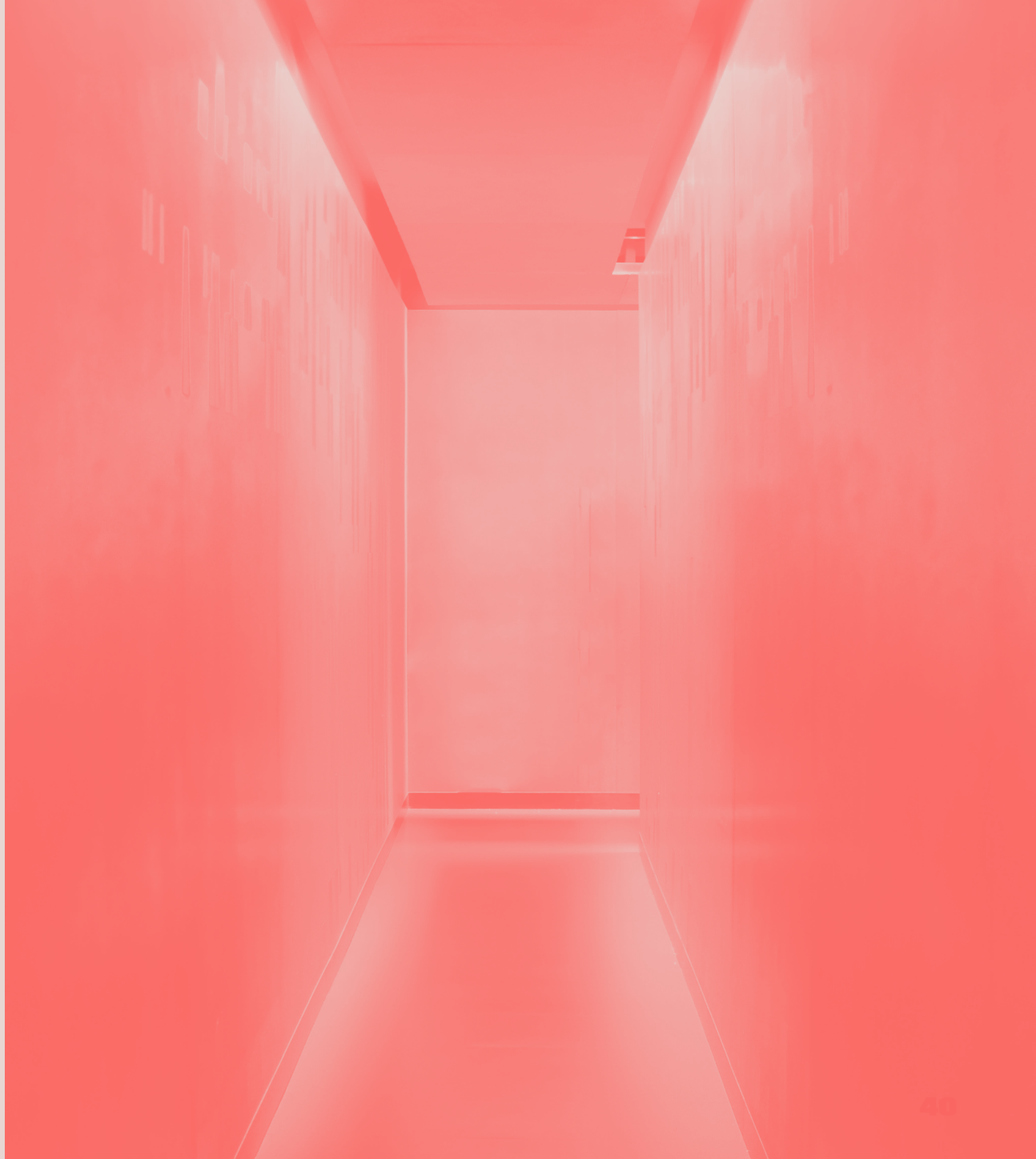
EXERCISE 2

- WITH more than 3 assigness,
  - only display 3 assignees
  - display a show more button



EXERCISE 3

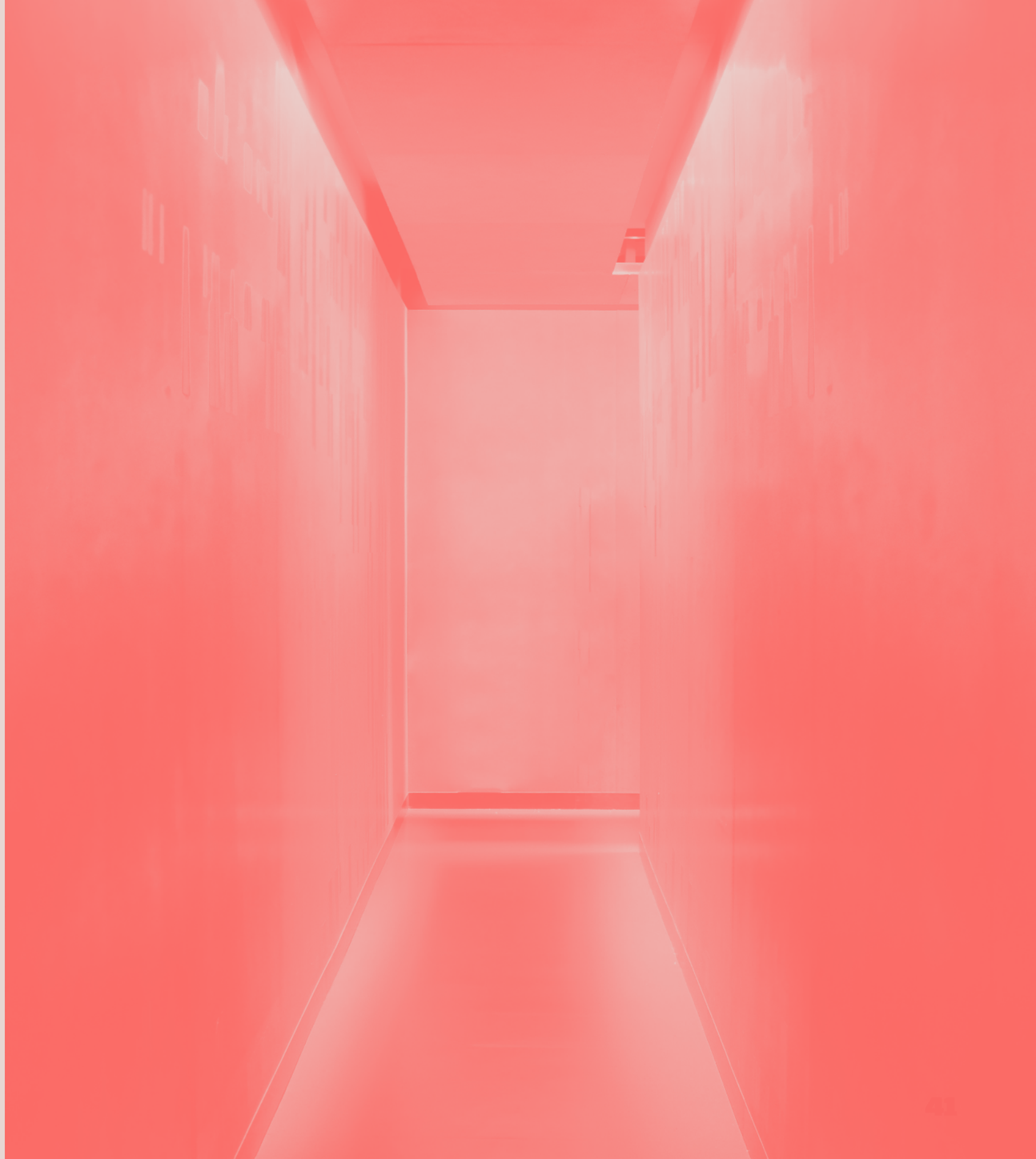
- WHEN the show more button was clicked
  - display all assignees
  - a show less button is displayed instead of a show more button
  - AND the show less button was clicked, only displays 3 assignees



EXERCISE 4

» WITH less than 4 assignees,

  » don't display a "show more" button



FEEDBACK

» Questions: tmayrhofer.lba@fh-salzburg.ac.at

» https://s.surveyplanet.com/x1ibwm85