
consider
commit
difference
tests
unit
run
xunit
testing
programmers
test
suite
code
class
something
enough
working
good
Indeed
isolation
frequently
often
think
collaborators
also
single
part
seconds
time
might
due
like
software
using
quicky
slow
term
common
people
Kent
discussion
least
classes
style
price
view
compile
remote
include
compiling
development
takes
testers
treat
system
talking
faster
non-determinism
classic
collaboration
Although
find
bugs
speed
rule
XP
write
change
Fowler
elements
look
know
useful
became
programmer
different
external
real
fault
one
really
xunit
Intermediate Unit Tests
Bob Fornal and Luis Carmona

About Me

- Husband and Father
- Senior Solutions Developer at Leading EDJE, Inc.

- 1978 - Started Writing Code
- 1990 - Completed BS CPS

About Luis

- Software Engineer at #100Devs

- 2019 - Built my first PC
- 2022 - Started writing code
- 2022 - Co-host of The Boba and Cherry Cafe twitter space.

Unit Testing Goals

- To have **CONFIDENCE** in code being released without defects.
- And **PROOF** that it works.
- Be **PRACTICAL** about how we test;
- Be **PRUDENT** about what should be tested.
- To **DECREASE** defects (find BUGS earlier) getting to QA.
- And **FASTER**, **EASIER** deployments.

What is Unit Testing?

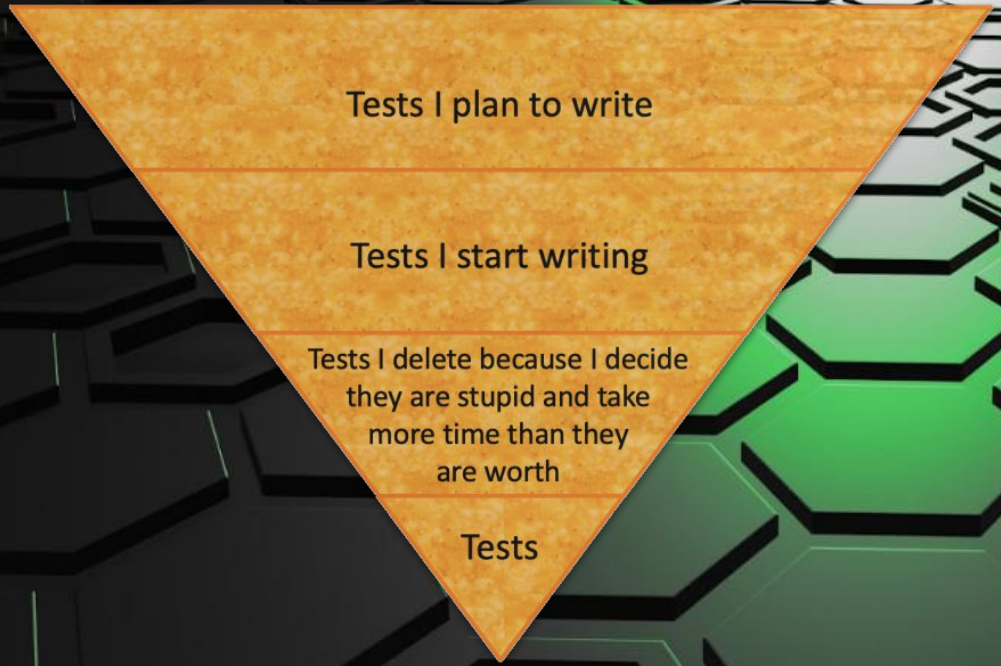
- ATOMIC, Lowest Level (small and fast).
- Single Responsibility Principle (SRP): “do one thing well.”
- Repeatable, Reliable, and Deterministic.
- Demonstrate concrete progress.
- Fails on a bug or changed requirements.
- Easy to understand why it fails.
- Reduce the cost of bugs.

Overview

- Stubs
- Spies
- Mocks
- Dummies

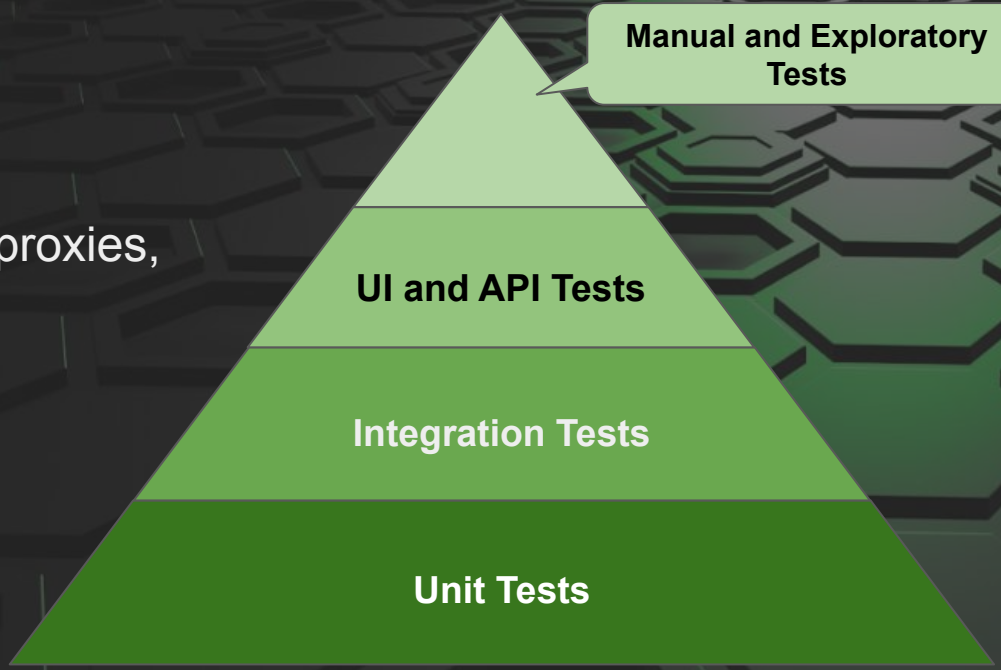
- Code Coverage

- Asynchronous Activity
- Dates

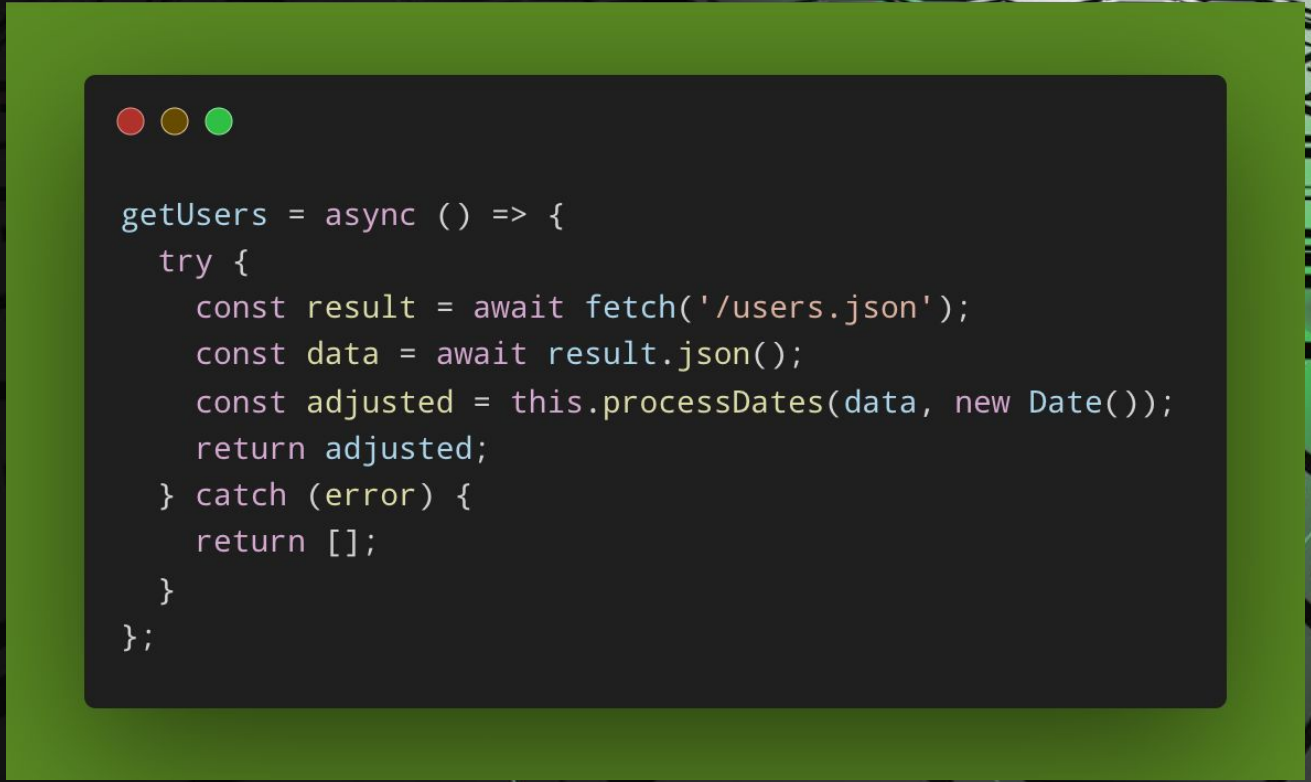


Definitions

- System Under Test:
  - Subject Under Test (SUT)
  - Code Under Test (CUT)
- Dependency
- Mocking: test double, stubs, proxies, mocks, or spies.

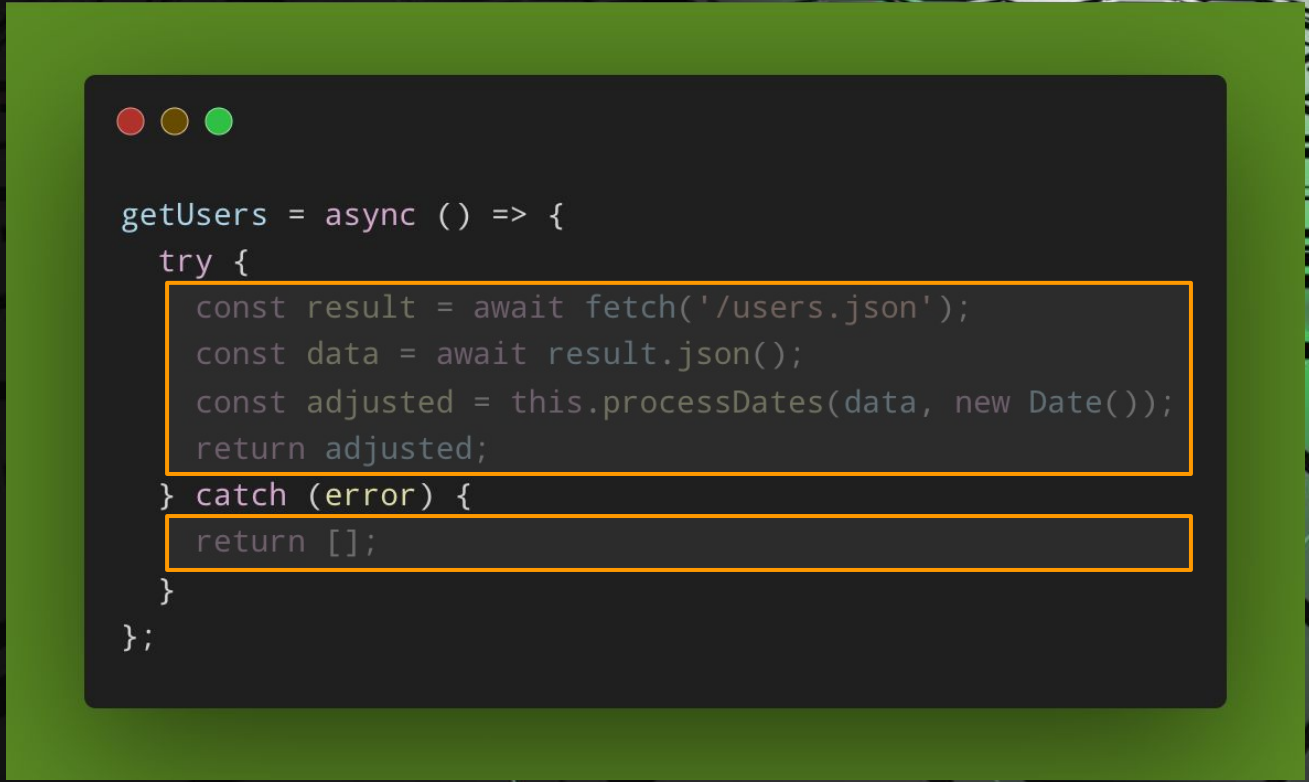


Anatomy of Code Under Test

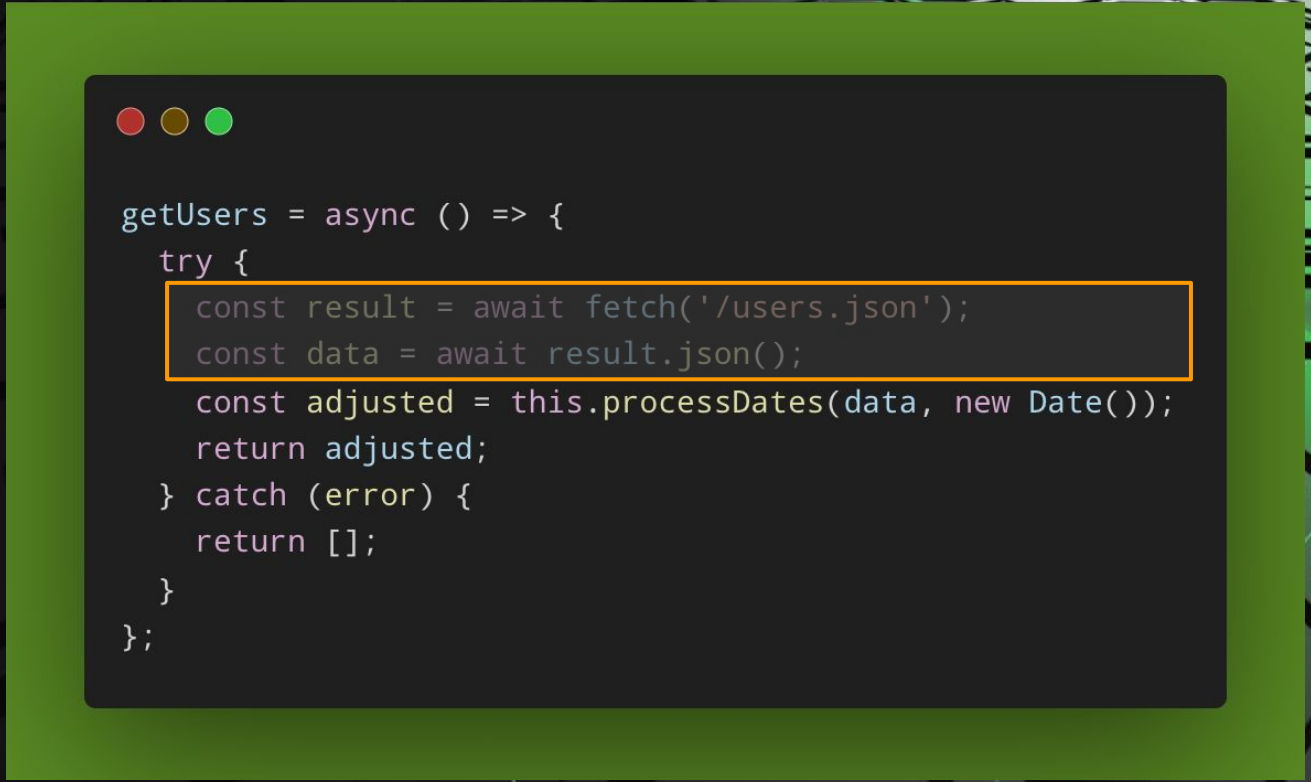


```
getUsers = async () => {
  try {
    const result = await fetch('/users.json');
    const data = await result.json();
    const adjusted = this.processDates(data, new Date());
    return adjusted;
  } catch (error) {
    return [];
  }
};
```

Anatomy of Code Under Test



Anatomy of Code Under Test



```
getUsers = async () => {
  try {
    const result = await fetch('/users.json');
    const data = await result.json();
    const adjusted = this.processDates(data, new Date());
    return adjusted;
  } catch (error) {
    return [];
  }
};
```

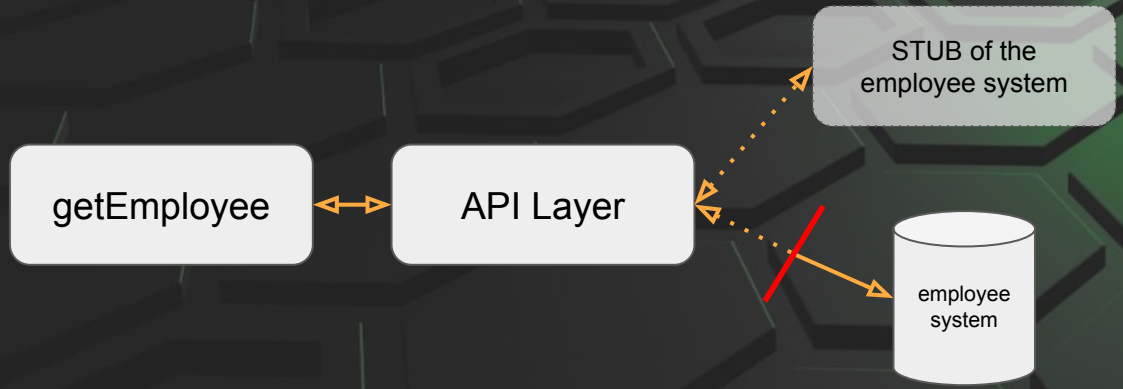
Anatomy of Code Under Test

```
getUsers = async () => {
  try {
    const result = await fetch('/users.json');
    const data = await result.json();
    const adjusted = this.processDates(data, new Date());
    return adjusted;
  } catch (error) {
    return [];
  }
};
```

Stubs

The STUB returns a known and controlled value.

- Replace or redefine a function or method.
- Helps deal with interaction with outside code.



Stubs: Code

```
let count = 0;
const counter = {
  increment: () => count += 1,
  getCount: () => count
};
const app = counter => counter.increment();

describe('Counter Stubs', () => {
  it('expects app with mock counter .toHaveBeenCalledTimes', () => {
    const mockCounter = {
      increment: jest.fn()
    };

    app(mockCounter); // count does not get incremented
    expect(mockCounter.increment).toHaveBeenCalledTimes(1);
    expect(counter.getCount()).toEqual(0);
  });
});
```

Stubs: Code

```
let count = 0;
const counter = {
  increment: () => count += 1,
  getCount: () => count
};
const app = counter => counter.increment();

describe('Counter Stubs', () => {
  it('expects app with mock counter .toHaveBeenCalledTimes', () => {
    const mockCounter = {
      increment: jest.fn()
    };

    app(mockCounter); // count does not get incremented
    expect(mockCounter.increment).toHaveBeenCalledTimes(1);
    expect(counter.getCount()).toEqual(0);
  });
});
```


CODE

Spies

The **SPY** are Stubs that gather execution information.

- Did the spied on method/function get called, how many times, when, and with what parameters.
- Unlike Stubs, a **SPY** wraps the target method/function instead of replacing it, so the original code of the target can also be executed.

Spies: Code

```
class MethodClass {
  instanceMethod = () => 'output from instance method';
}

describe('Spying 001', () => {
  let service;

  beforeEach(() => {
    service = new MethodClass();
    jest.clearAllMocks();
  });

  it('expects to spy on instance method and change implementation', () => {
    jest.spyOn(service, 'instanceMethod').mockImplementation(() => 'spy triggered');

    const result = service.instanceMethod();
    expect(service.instanceMethod).toHaveBeenCalledTimes(1);
    expect(result).toEqual('spy triggered');
  });
});
```

Spies: Code

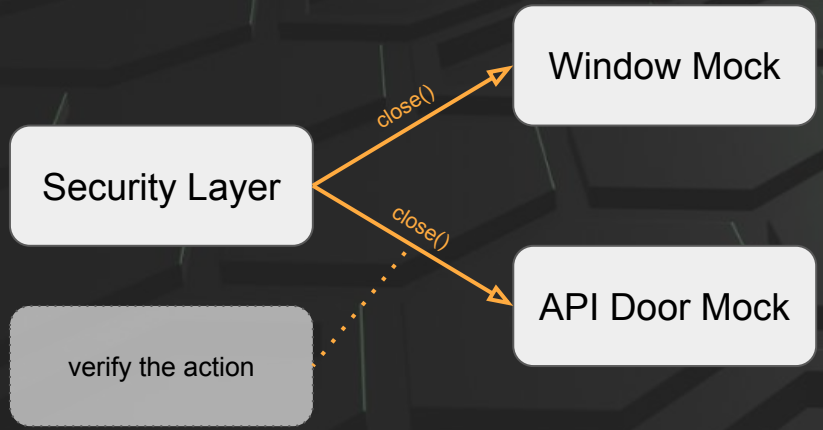
```
class MethodClass {
  instanceMethod = () => 'output from instance method';
}

describe('Spying 001', () => {
  let service;

  beforeEach(() => {
    service = new MethodClass();
    jest.clearAllMocks();
  });

  it('expects to spy on instance method and change implementation', () => {
    jest.spyOn(service, 'instanceMethod').mockImplementation(() => 'spy triggered');

    const result = service.instanceMethod();
    expect(service.instanceMethod).toHaveBeenCalledTimes(1);
    expect(result).toEqual('spy triggered');
  });
});
```


CODE

Mocks

The MOCK sets the expected behavior(s) on real objects or functions.

- *Like the twin brother to Stubs.*
- This does not replace functionality, just changing its behavior in specific cases.



Mocks: Code

```
function forEach(items, callback) {
  for (let data of items) {
    callback(data);
  }
}

describe('Testing Mocks 003', () => {
  it('expects the use of the .mock property', () => {
    const mockCallback = jest.fn(x => 42 + x);

    forEach([0, 1], mockCallback);
    expect(mockCallback.mock.calls.length).toBe(2);
    expect(mockCallback.mock.calls[0][0]).toBe(0);
    expect(mockCallback.mock.calls[1][0]).toBe(1);
    expect(mockCallback.mock.results[0].value).toBe(42);
  });
});
```

Mocks: Code

```
function forEach(items, callback) {
  for (let data of items) {
    callback(data);
  }
}

describe('Testing Mocks 003', () => {
  it('expects the use of the .mock property', () => {
    const mockCallback = jest.fn(x => 42 + x);

    forEach([0, 1], mockCallback);
    expect(mockCallback.mock.calls.length).toBe(2);
    expect(mockCallback.mock.calls[0][0]).toBe(0);
    expect(mockCallback.mock.calls[1][0]).toBe(1);
    expect(mockCallback.mock.results[0].value).toBe(42);
  });
});
```


CODE

Dummies

The **DUMMY** is a simple object that serve no real purpose other than being available when the syntax requires it.

- Strongly typed languages.


CODE

Code Coverage

- Statements
- Branches
- Functions
- Lines

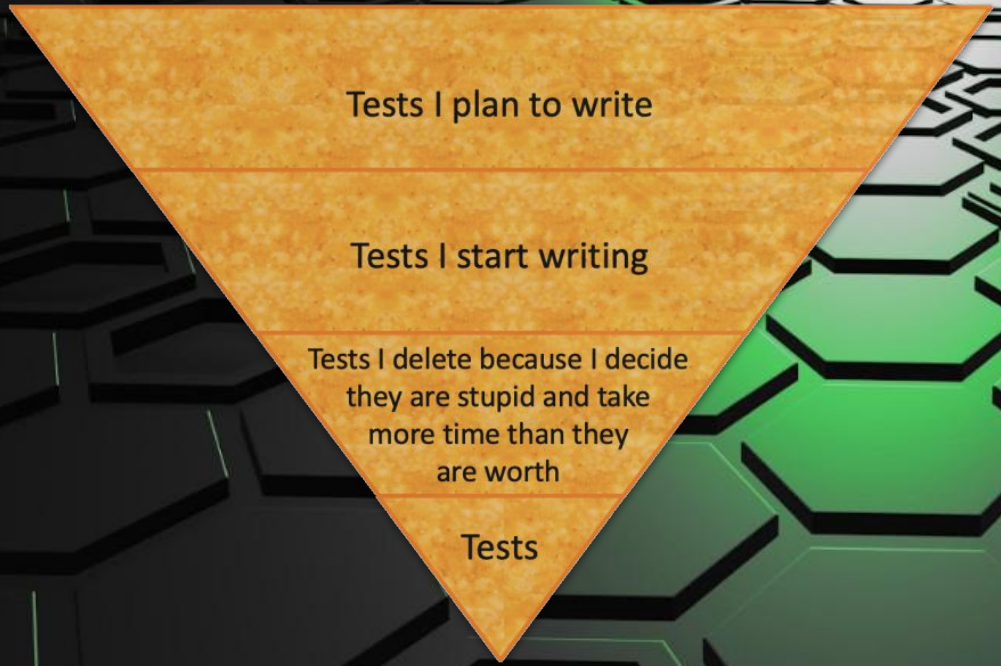
DEMO

Summary

- Stubs
- Spies
- Mocks
- Dummies

- Code Coverage

- Asynchronous Activity
- Dates




consider
commit
difference
tests
unit
run
xunit
testing
programmers
test
suite
code
class
something
enough
working
good
Indeed
isolation
frequently
often
think
collaborators
also
single
part
seconds
time
might
due
like
software
using
quicky
slow
term
common
people
Kent
discussion
least
classes
style
price
view
compile
remote
include
compiling
development
takes
testers
treat
system
talking
faster
non-determinism
classic
collaboration
Although
find
bugs
speed
rule
XP
write
change
Fowler
elements
look
know
useful
became
programmer
different
external
real
fault
one
really
xunit
Intermediate Unit Tests
Bob Fornal and Luis Carmona