Testing for the *skeptical* developer

Based on a presentation by Andrei Sebastian Cîmpean
Common excuses

➔ It’s slowing me down...
➔ It wasn’t done from the start of the project...
➔ The client doesn’t pay for it...
➔ It’s not worth it / I’m testing my code before push
➔ We have QA

➔ IT’S HARD
➔ I’M AFRAID
Different levels of Software Testing

1. **Unit Testing** *(required)*: Individual units/components of a software/system are tested. Validate that each unit of the software performs as designed.

2. **Integration Testing** *(required)*: Individual units are combined and tested as a group. Expose faults in the interaction between integrated units.

3. **System Testing**: A level of the software testing process where a complete, integrated system/software is tested. The purpose of this test is to evaluate the system’s compliance with the specified requirements.

4. **Acceptance Testing**: A level of the software testing process where a system is tested for acceptability. The purpose of this test is to evaluate the system’s compliance with the business requirements and assess whether it is acceptable for delivery.
Different Levels of Software Testing

**Unit Testing**
(Do the parts perform correctly alone?)

**Integration Testing**
(Do the parts perform correctly together?)

**System Testing**
(Does the complete integrated system comply with the specs?)

**User Acceptance or End-to-End (e2e) Testing**
(Does the system meet the end user’s expectations?)
Test doubles

“It’s hard”
“It’s slowing me down”

➔ Dummy
➔ Fake
➔ Stubs
➔ Spies
➔ Mocks
**Dummy** objects are passed around but never actually used (but are required to compile).

**Fake objects** actually have working implementations, but usually take some shortcut which makes them not suitable for production.

**Stubs** provide canned answers to calls made during the test. Stubs may also record information about calls.

**Spies** are stubs that gather execution information, so they can tell you, in the end, what got called, when and with which parameters.

**Mocks** are objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

[https://blog.bitsrc.io/unit-testing-deep-dive-what-are-stubs-mocks-spies-and-dummies-6f7fde21f710](https://blog.bitsrc.io/unit-testing-deep-dive-what-are-stubs-mocks-spies-and-dummies-6f7fde21f710)
Stone Age Practice: Traditional QA

QA Testing Establishes Tests That Typically Measure Software’s Adherence to Documented Requirements
Acceptance Test-Driven Development establishes a feedback loop that assures adherence to customer requirements.
Behavior-Driven Development codifies customer requirements into scenarios that are automated in a TDD test-first iteration and output test reports as part of a continuous build integration cycle.
Even More Modern Recipe: Behavior-Driven Development

Behavior-Driven Development codifies customer requirements into discrete, real-life scenarios that are automated in testing frameworks and output test reports as part of a continuous integration pipeline.
Scrum
Test Driven Development

➔ Breaking a problem up into smaller pieces
➔ Defining the “simplest thing that could possibly work”
➔ Improved design
It’s not necessary to write unit tests to do TDD.

It’s not necessary to do TDD to write unit tests.
What is TDD according to Wikipedia

A development process that relies on the repetition of a very short development cycle:

1. The developer writes an (initially failing) automated test case that defines a desired improvement or new function

2. The developer produces the minimum amount of code to pass that test

3. The developer refactors the new code to acceptable standards
Write failing test

```javascript
// tests/part1/cart-summary-test.js
var chai = require('chai');
var expect = chai.expect; // we are using the "expect" style of Chai
var CartSummary = require('./../../../src/part1/cart-summary');

describe('CartSummary', function() {
  it('getSubtotal() should return 0 if no items are passed in', function() {
    var cartSummary = new CartSummary([]);
    expect(cartSummary.getSubtotal()).to.equal(0);
  });
});
```

from https://www.codementor.io/nodejs/tutorial/unit-testing-nodejs-tdd-mocha-sinon
Write failing test

```javascript
CartSummary
  1) getSubtotal() should return 0 if no items are passed in

0 passing (7ms)
1 failing

1) CartSummary getSubtotal() should return 0 if no items are passed in
   TypeError: CartSummary is not a function
   at Context.<anonymous> (tests/cart-summary-test.js:8:21)
```

from https://www.codementor.io/nodejs/tutorial/unit-testing-nodejs-tdd-mocha-sinon
Write just enough code to pass the test

```javascript
// src/part1/cart-summary.js
function CartSummary() {

    CartSummary.prototype.getSubtotal = function() {
        return 0;
    }

    module.exports = CartSummary;

from https://www.codementor.io/nodejs/tutorial/unit-testing-nodejs-tdd-mocha-sinon
```
Write just enough code to pass the test

```
CartSummary
✓ getSubtotal() should return 0 if no items are passed in

1 passing (1ms)
```

from https://www.codementor.io/nodejs/tutorial/unit-testing-nodejs-tdd-mocha-sinon
Write failing test

```javascript
it('getSubtotal() should return the sum of the price * quantity for all items',

    var cartSummary = new CartSummary([{
        id: 1,
        quantity: 4,
        price: 50
    }, {
        id: 2,
        quantity: 2,
        price: 30
    }, {
        id: 3,
        quantity: 1,
        price: 40
    }]);

    expect(cartSummary.getSubtotal()).to.equal(300);
});
```

from https://www.codementor.io/nodejs/tutorial/unit-testing-nodejs-tdd-mocha-sinon
Write failing test

```javascript
) CartSummary getSubtotal() should return the sum of the price * quantity for all item:

AssertionError: expected 0 to equal 300
+ expected - actual

-0
+300

at Context.<anonymous> (tests/cart-summary-test.js:30:40)
```

from https://www.codementor.io/nodejs/tutorial/unit-testing-nodejs-tdd-mocha-sinon
Write just enough code to pass the test

// src/part1/cart-summary.js
function CartSummary(items) {
  this._items = items;
}

CartSummary.prototype.getSubtotal = function() {
  if (this._items.length) {
    return this._items.reduce(function(subtotal, item) {
      return subtotal += (item.quantity * item.price);
    }, 0);
  }
  return 0;
};

from https://www.codementor.io/nodejs/tutorial/unit-testing-nodejs-tdd-mocha-sinon
Specific tests lead to generic production code

Every new test case makes the test suite more constrained and more specific.

Just adding if statements **will not work** in the long run. Developers end up making tests pass by innovating general algorithms.

To make the new test case pass, the programmer strives to make the production code more general, not more specific.
→ Programmers deserve to feel confident that their code works and TDD is one (not the only) way to reach that

→ Tackle problems piecemeal, figuring out specific cases without having to solve the general case all at once

→ Leads to more modularized, flexible, and extensible code

→ Existing tests provide a safety net

→ Confusion over the definition of TDD and unit testing

→ Many people make bad trade-offs, especially with heavy mocking that end up driving architecture

→ “HARDER”
→ TDD gives a mechanism to quickly get feedback on an idea and example usage of an API while implementing it one step at a time.

→ Can be very appealing to developers who can find a large problem overwhelming

→ Programmers are able to focus exclusively on the task at hand: make the individual test pass

→ TDD is may leave you uncertain what exactly to do with the **Refactor** phase.
Behaviour Driven Development

➔ An executable specification that fails because the feature doesn't exist

➔ Writing the simplest code that can make the spec pass.

➔ Repeat this until a release candidate is ready to ship.

➔ Easier to read TDD?
Behavior-driven development is facilitated by a simple domain-specific language (DSL) using natural language constructs (e.g., English-like sentences) that can express the behavior and the expected outcomes.

BDD suggests that unit test names be whole sentences starting with a conditional verb ("should" in English for example).

BDD suggests that unit tests should be written in order of business value.

from https://en.wikipedia.org/wiki/Behavior-driven_development#Behavioural_specifications
Feature: Belly

Scenario: a few cucues
  Given I have 42 cucues in my belly
  When I wait 1 hour
  Then my belly should growl

file: StepDefinitions.java

```java
package se.thinkcode.ike;

import cucumber.api.java.en.Given;
import cucumber.api.java.en.Then;
import cucumber.api.java.en.When;

import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;

public class StepDefinitions {
  private Belly belly;
  private int waitingTime;

  @Given("^I have \d+ cucues in my belly$")
  public void i_have_cucues_in_my_belly(int cucues) throws Throwable {
    belly = new Belly();
    belly.eat(cucues);
  }

  @When("^I wait \d+ hours$")
  public void i_wait_hour(int waitingTime) throws Throwable {
    this.waitingTime = waitingTime;
  }

  @Then("my belly should \w+$")
  public void my_belly_should_growl(String expectedSound) throws Throwable {
    String actualSound = belly.getSound(waitingTime);
    assertThat(actualSound, is(expectedSound));
  }
}
```

from https://thomassundberg.wordpress.com/2014/05/29/cucumber-jvm-hello-world/
So if we write tests in a BDD style ...

1. Customers can read them
2. Customers can give us early feedback
3. Customers can write some of them
So if we write tests in a BDD style ... 

1. Customers can read them
2. Customers can give us early feedback
3. Customers can write some of them

This never* happens
Since BDD is basically a feature driven approach to TDD, there is value in doing it if you want to.
Common excuses again...

➔ It’s slowing me down...
➔ It wasn’t done from the start of the project...
➔ The client doesn’t pay for it...
➔ It’s not worth it / I’m testing my code before push
➔ We have QA

➔ IT’S HARD
➔ I’M AFRAID