

W4156

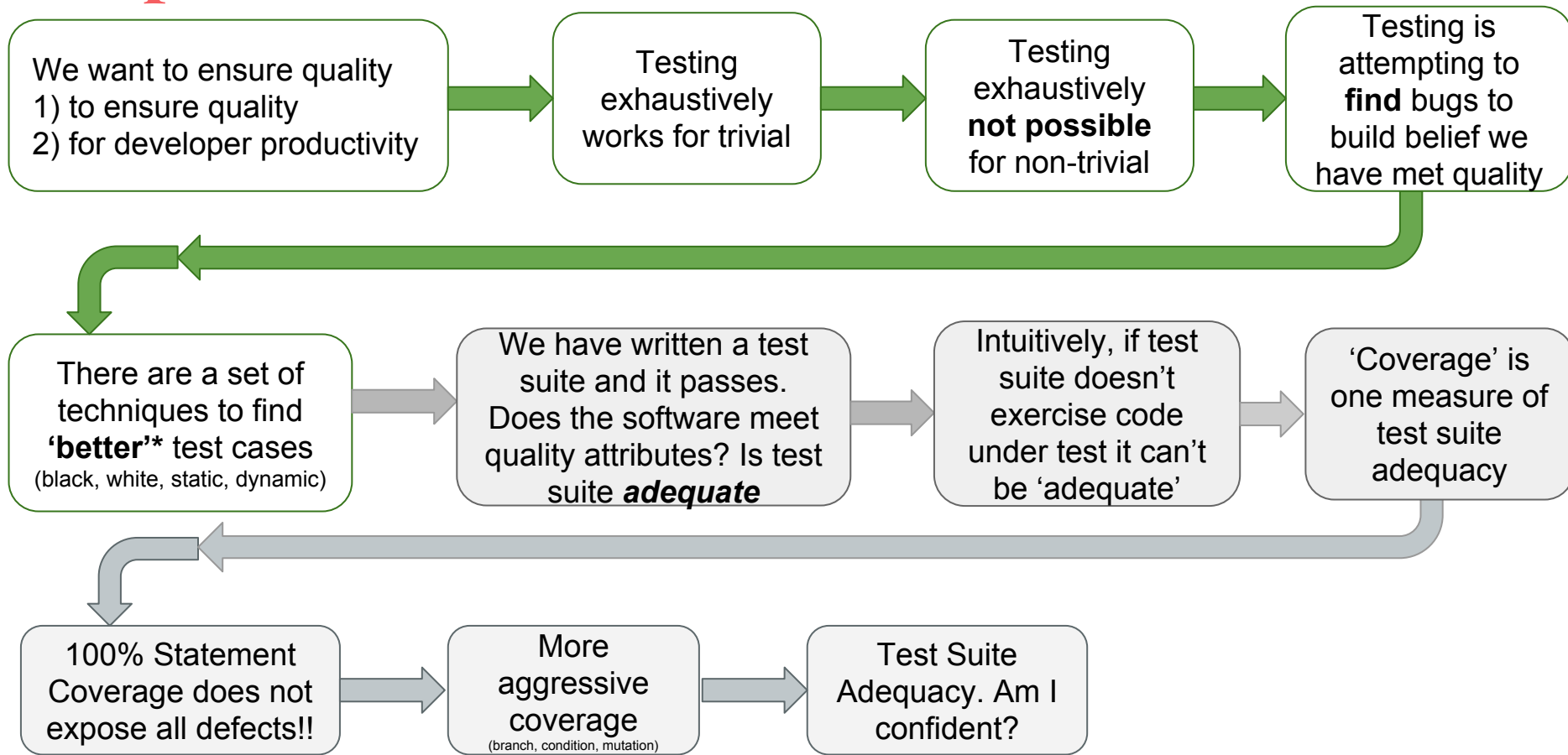
Testing III

Disclaimer

Note:

- There are many software development methodologies and techniques
- The course goal is to teach **theory, practice** and **judgement**
- We will teach techniques/methodologies as exemplars
- Students will learn to apply to different situations:
 - Pros and cons
 - When to apply techniques/methodologies / when to customize
 - Remember – teams work in different industries, problem domains, and maturity (the way you develop/test a plane != search engine != trading platform != etc)
- We are going to cover TDD (it inspired strong reactions). I want to teach so you can understand and incorporate into your ‘toolkit’

Recap



Agenda

Today we are going to apply theory to software development process/practice

- ❑ Testing aiding *productivity*
- ❑ Testing as part of the SDLC (TDD)
- ❑ Mocking
- ❑ 'Levels' of Testing

Why Test II

On the train from NY to Philly I got into conversation with a entrepreneur

He was building an app for Veterinarians*.

He was super excited and talking about his company.

But as we got deeper into the conversation he revealed a source of stress:

- As the company grew customers complexity grew
- Team could not build/release new features without breaking existing functionality
- Either
 - Don't release features and don't acquire new customers
 - Release features and upset existing customers
- The overall pace of development was also slowing!
- Customers were getting unhappy

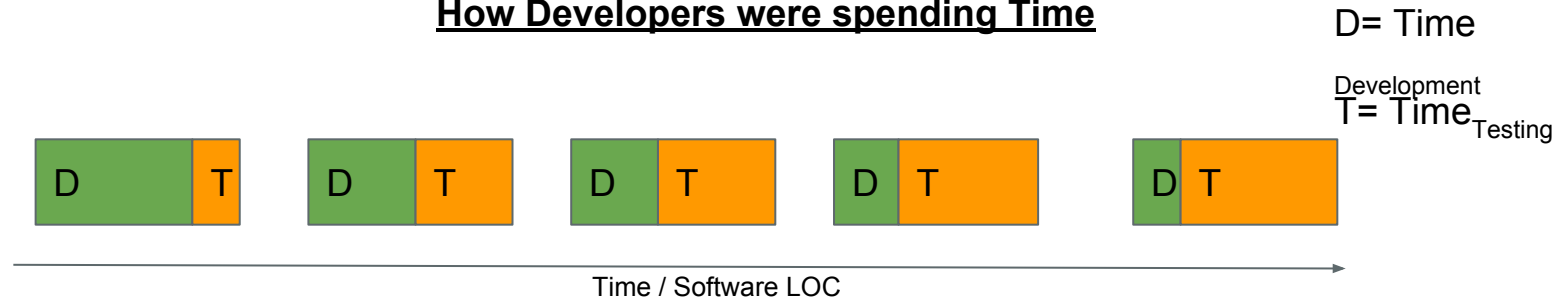
Why was this happening?

* industry changed to protect the innocent / company is google-able

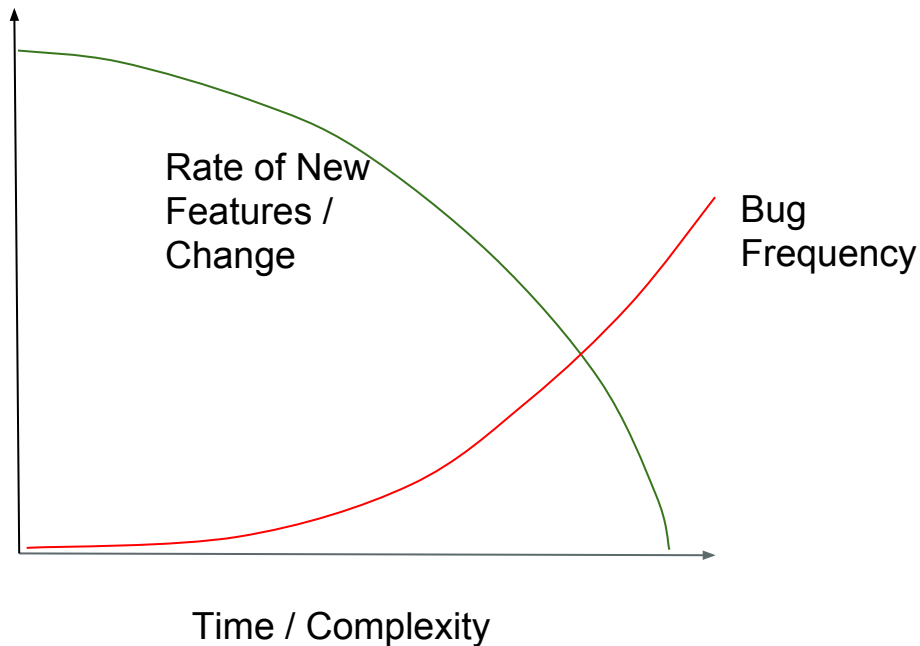
What was going on?

1. As the complexity of the product grew there were more features / lines of code.
2. Developers were **manually** testing the software (click, click, check, etc)
3. As complexity grew they spent more time **manually testing** vs **developing** new features
4. Testing was **repeated** each release (they re-executed the same test cases each month)
5. Manually testing is also **error prone** so bugs leaked through
6. Furthermore, when bugs occurred they were **hard to trace**

How Developers were spending Time



Poor Testing Practices vs Complexity



Remember: software is the codification of a business (process, rules, etc).
If you can't change the software you can't change the business. Over time someone will do what you wanted to

Differential Diagnosis and Treatment

Diagnosis

- No *automated* testing / generally poor testing practices
- Reduced developer productivity
- Only going to get worse (sometimes called the 'death spiral of IT')

Treatment Plan

The hard fact is there is technical *debt*

(it will take time to build automated tests and generally you can't stop for a month)

- Change team *culture* to value automated testing (and writing 'good' test cases)
- Introduce *incrementally* to address highest risk / damage
- Repeat
- Climb out of the hole slowly
- N months time this problem *could* be conquered
(or you will keep pace with complexity)

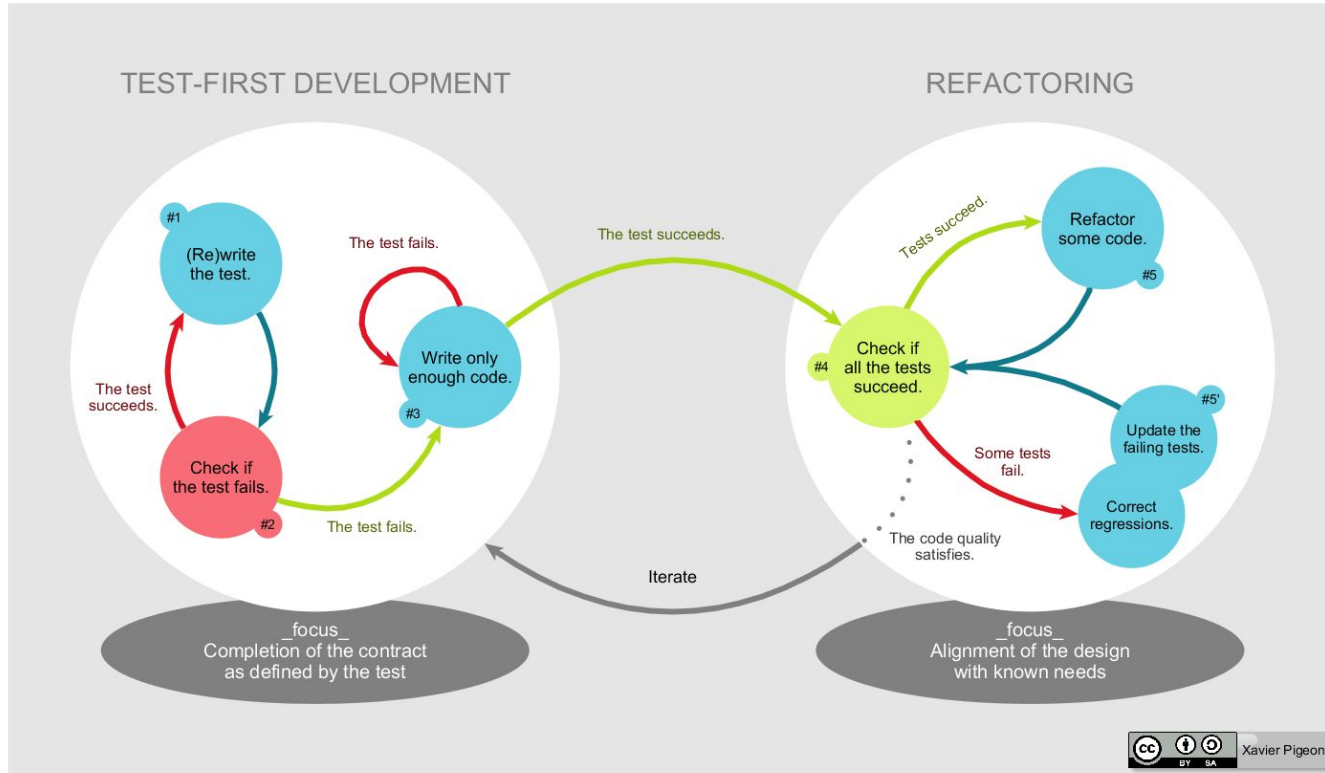
Test Driven Development

We saw in our tale about testing productivity (the vet software) that without automated testing the rate of change can slow.

1. What if we wrote the tests as we wrote the software?
2. Would we write tests as we ‘go along’?
3. Could this even help developers be more productive?
4. Could writing the tests as we write code ensure we write testable code?*

* To build associations. Remember testability is an internal quality attribute (we covered this in requirements)

Test Driven Development



TDD: Example

80% of teams each year end up writing some code that “validates” a URL

For their purposes “valid” means

- URL is well formed
- Is not local
- Exists/is real
- Site is non-empty

(Remember: don't write foundational code. Borrow libraries)

Step 0: Think

1. *Analyze* the problem you are about to solve
2. Think about the functionality?
3. What are the test cases?

Step 1: Write Test

#1

(Re)write
the test.

```
def test_url_validator(self):
    cases = [("", ValueError),
              ("http://www.google.com", True),
              ("https://www.google.com/", True),
              ("clearlynotvalid", ValueError),
              ("http://validbutnocontent.com", False)]

    for c in cases:
        self.push_assert(c[0], c[1])
```

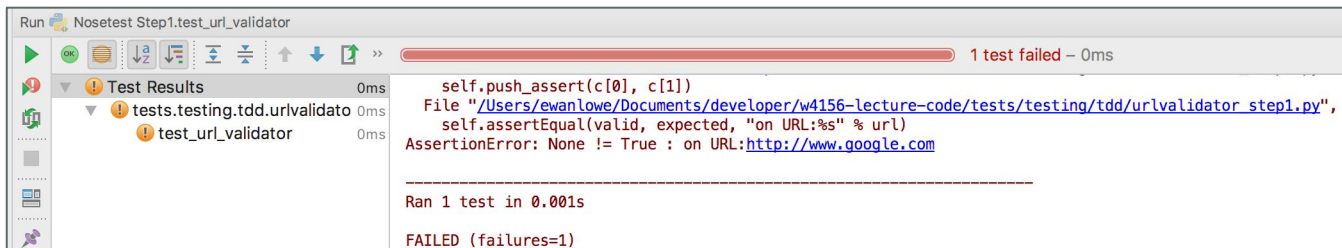
```
class URLValidator:

    def validate_url(self, url: str) -> bool:
        pass
```

Step 2: Run Tests

Check if
the test fails.

#2



```
Run Nostest Step1.test_url_validator

Test Results
  tests.testing.tdd.urlvalidato
    test_url_validator

self.push_assert(c[0], c[1])
File "/Users/ewanlowe/Documents/developer/w4156-lecture-code/tests/testing/tdd/urlvalidator_step1.py",
self.assertEqual(valid, expected, "on URL:%s" % url)
AssertionError: None != True : on URL:http://www.google.com

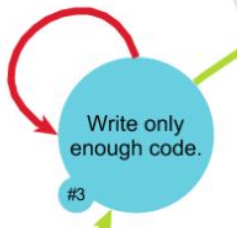
Ran 1 test in 0.001s

FAILED (failures=1)
```

In the first example we are writing new code. However, on the second cycle we may be adding functionality to code that already exists. Therefore we can write tests for new functionality. Generally it doesn't, but there is a non-zero probability the tests can pass/functionality exists

Step 3a: Write Enough Code

The test fails.



```
class URLValidator:

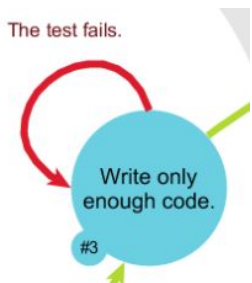
    @staticmethod
    def validate_url(url: str) -> bool:
        syntactically_well_formed = validators.url(url)

        if not syntactically_well_formed:
            raise ValueError("Must supply well formed URL")

        r = requests.get(url)
        # We will only consider OK as valid. 204 is 'invalid' from our perspective
        if r.status_code == 200 and len(r.text) > 0:
            return True
        else:
            return False
```

Step 3b: Write Enough Code

Amusingly (and in line with TDD) while writing this example I had an unanticipated scenario.



The ISP in Arizona ‘nicely’ created a valid response page saying in HTML “I am sorry. We couldn’t find that page”. This caused the last test case to fail.

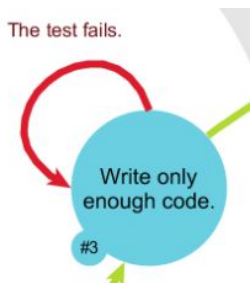
```
def test_url_validator(self):
    cases = [("", ValueError),
              ("http://www.google.com", True),
              ("https://www.google.com/", True),
              ("clearlynotvalid", ValueError),
              ("http://validbutnocontent.com", False)]

    for c in cases:
        self.push_assert(c[0], c[1])
```

ISP returned a page
causing this assertion
to fail

Step 3c: Write Enough Code

But that is ok – we now write more code to fix this scenario



```
class URLValidator:
    error_string = "http://finder.cox.net/"

    def validate_url(self, url: str) -> bool:
        syntactically_well_formed = validators.url(url)

        if not syntactically_well_formed:
            raise ValueError("Must supply well formed URL")

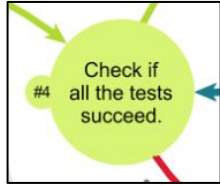
        r = requests.get(url)

        # We will only consider OK as valid. 204 is 'invalid' from our perspective
        if r.status_code == 200 and len(r.text) > 0 and (r.text.find(URLValidator.error_string) == -1):
            return True
        else:
            return False
```

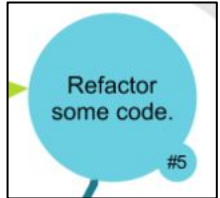
Re-run tests and they pass

(would probably write a more robust solution to 'whois' domain then hitting to ensure the server responds but suffices for exemplar)

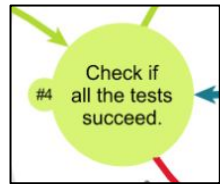
Closing the TDD Iteration



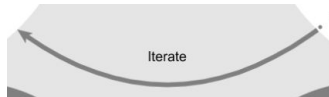
We would run entire test suite (in this example it was a fresh codebase)



Don't forget. If you want to refactor / clean up the code base you can do it now (we will cover refactoring in more detail in a later lecture). In this example I pulled out a variable, etc



Re-run tests



Onto the next piece of functionality. Rinse and repeat

Mocking

When we move beyond a single method or class we find dependencies between object/units.

How can we test a particular scope *in isolation* of other components?

Mocking: The isolation of a unit of code under test by placing ‘stubs’ or ‘mocks’ of dependent components to simulate their behavior. This allows us to independently test sub-components of our system.

Mocking Example

See `w4156-lecture-code`

Test Driven Development Pros and Cons

Pros	Cons
Encourages structured thinking at design time (requirements/specification and test cases)	Potential to make the design overly complex with additional interfaces
Ensures that the code design is 'testable' / decoupled components	Tests are created by developer therefore any assumptions may be baked into tests and code (does not obviate the need for an distinct tester)
Ensures tests are created/ Tests are part of development (which often gets deprioritized)	Testing takes time and is code that needs to be maintained (we are really arguing whether testing is net positive)
Saves time individually and as a team / accelerates velocity	

Judgement

- We know that quality attributes impact design (testability is one).
 - Is the tradeoff for testability worth it?
- Is TDD appropriate in all scenarios?
- Is my test suite/approach adequate?

Judgement. Judgement. Judgement

I want you to do three things

1. Understand the techniques and the pros/cons
2. Understand the problem you are facing, reflect and apply appropriately
3. Don't join binary flame wars when the answer may be “it depends”

Testing Levels

We have covered testing theory, TDD and Mocking

We feel confident in testing reasonably small programs (a handful of classes).

How do we test *systems* (multiple components and larger codebases?)

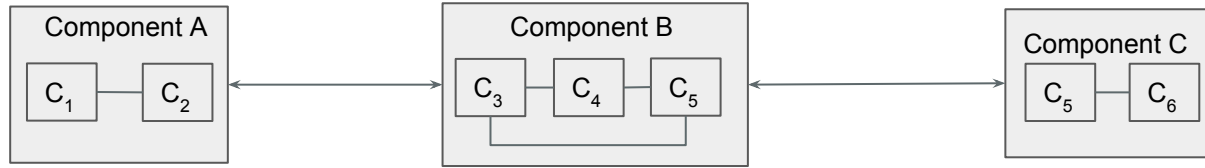
Testing Levels

C₅

Class



Scope of Test

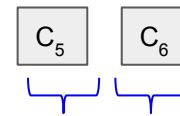
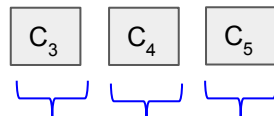
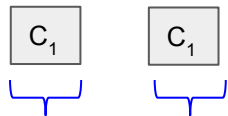


How do I test the overall system?

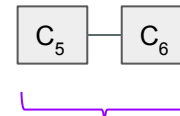
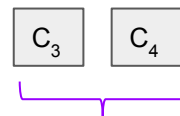
If I test C₁ and C₂ should I be confident the overall system works?

Testing in Industry (Simplified)

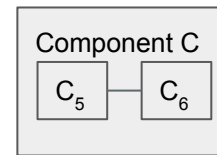
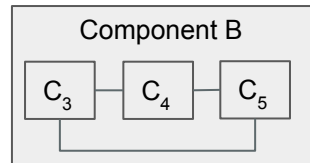
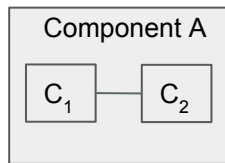
Unit Test



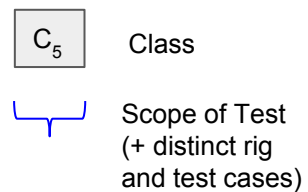
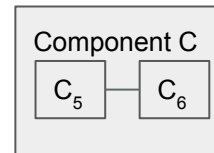
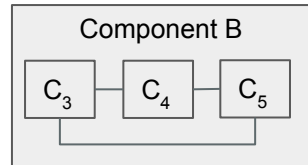
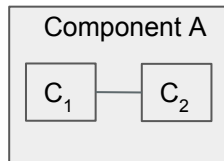
Integration
(Units)



Integration
(Component)



System



Levels of testing

Testing Level	Description
Unit	Individual Classes
Integration	A series of classes assembled into a module / system sub-function
System	The entire system assembled together

What are the pros/cons of each level?

Levels Example

See w4156-lecture-code



Project Recommendation

Being pragmatic: my recommendation given time and experience:

1. Try TDD for a couple of weeks

(you may get frustrated because TDD exposes that your designs are not testable yet (we will cover in later lectures) but view this positively - 'oh wow - trying to do TDD exposes the fact I write blobs etc')

2. Unit test any classes with the core of your business logic

3. Write flask-level tests to hit your APIs

4. Work out if it is worth mocking the database in #2 or #3

5. Apply judgement: is this design trade-off worth it?

Pop Quiz

Question	Our answer
Testing begins once the software is ready to ship <T/F>?	
The TDD order is <(re)write test, check if test fails, check if test passes, write code, refactor, return to start>?	
The purpose of a mock is to <>?	
If my unit tests pass then my system works?	
The different levels of testing are <conflicting / complimentary>	

Reading

Material	Optionality
Finish Beginning S.E. Chapter 4 (x-previous and this lecture)	Required
<u>Python mocking</u> (most languages have similar facilities)	Required