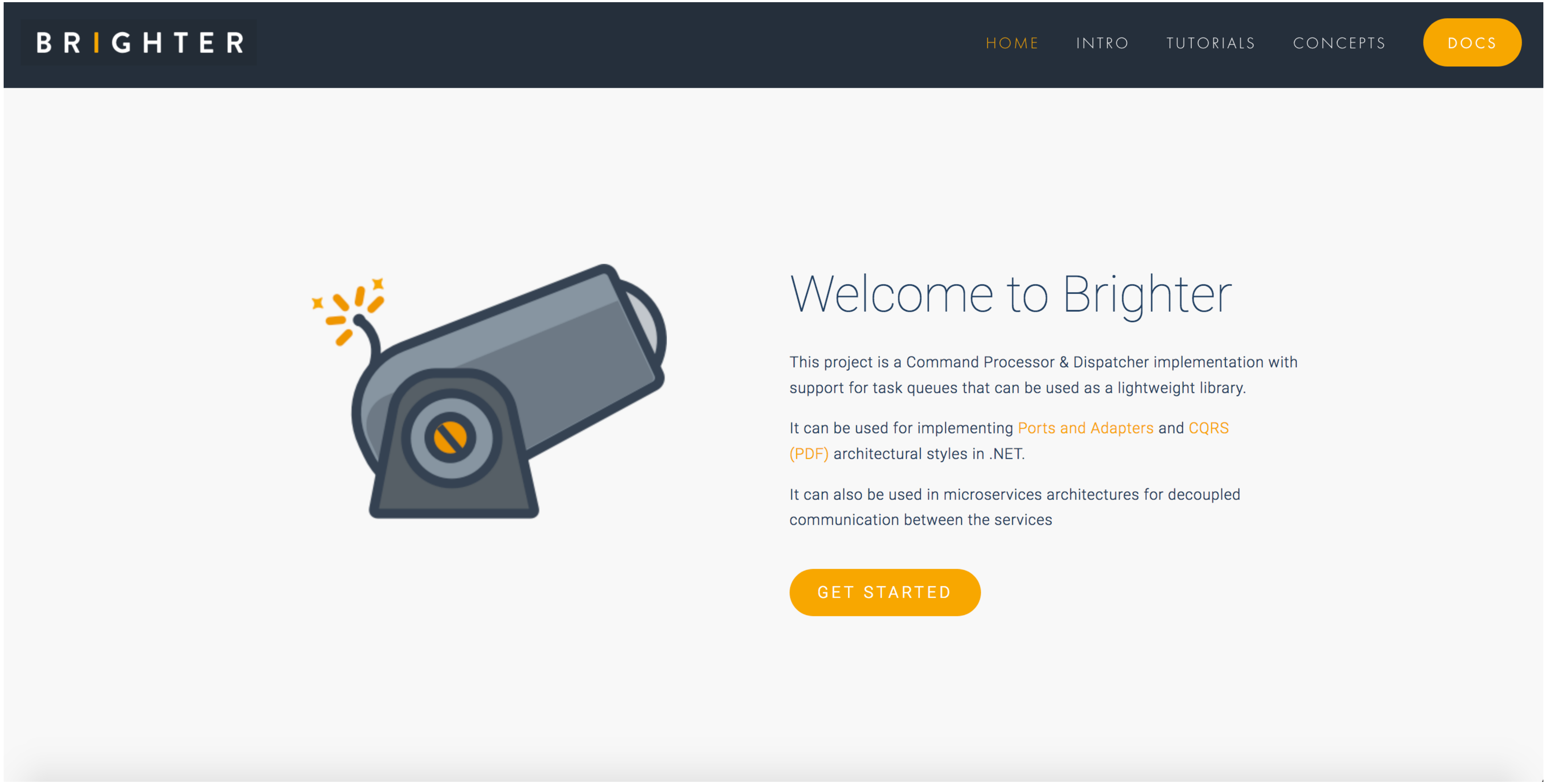
TDD Revisited

Where it all went wrong - guidance on what to do instead

Who are you?

- Software Developer for more than 20 years
  - Worked mainly for ISVs
    - Reuters, SunGard, Misys, Huddle
  - Worked for a couple of MIS departments
    - DTI, Beazley
- Microsoft MVP for C#
  - Interested in architecture and design
  - Interested in Agile methodologies and practices
- No smart guys
  - Just the guys in this room



Welcome to Brighter

This project is a Command Processor & Dispatcher implementation with support for task queues that can be used as a lightweight library.

It can be used for implementing Ports and Adapters and CQRS (PDF) architectural styles in .NET.

It can also be used in microservices architectures for decoupled communication between the services

Agenda

- The Fallacies of TDD
- Clean Architecture
- Summary

The Fallacies of TDD

Fallacy

1: Developers write Unit Tests

Definition

To unit test requires **test doubles**, it's how you isolate. The SUT must be able to replace any dependency with a test double (a **mock**)

To isolate issues that may arise, each test case should be tested independently. **Substitutes** such as method stubs, mock objects, fakes, and test harnesses can be used to assist testing a module in **isolation**.

In unit testing **isolation** becomes how approach testing. We isolate one SUT from another for **defect localization**. Originates with **modules** being separately tested.

https://en.wikipedia.org/wiki/Unit_testing

Belief

To do this, we either have to design our class hierarchy before writing a test, and know we will mock a call…

Need-driven Development [is a] variation on the test-driven development process where code is written from the **outside in** and all depended-on code is replaced by Mock Objects that verify the **expected indirect outputs** of the code being written.

The consequence here is that we must **understand the details** of the SUT not just the **contract**… the details are **coupled to our test**, we can't change them without **changing our tests**.

… or we need to stop when we hit something outside our **single responsibility** when implementing and replace it with a **test double**.

Meszaros, Gerard. xUnit Test Patterns

Experience

The DI requirements here lead us towards an IoC container over Poor Man's DI because we have a graph of dependencies to realize at runtime

In a strongly-typed language this means an **interface stands-in for the actual dependency** and must be **injected into our SUT**.

When I look around now, I see a lot of people using **mocks to replace all their dependencies**. My concern is that they will begin to hit the **Fragile Test** issues that mocks present. Gerard Meszaros identifies the issues we hit as two specific smells: **Overspecified Software** and **Behavior Sensitivity**.

Our tests should focus on the contract, but here they focus on the implementation, which makes them hard to read as there is a lot of setup code.

If we change how our code works, a lot of tests may break – we say that our tests are sensitive to changes in the details.

http://codebetter.com/iancooper/2007/12/19/mocks-and-the-dangers-of-overspecified-software/

Principle

1: Developers write Developer Tests

Observation

This is the only use of the phrase "unit test" in the book, Kent is referring here to his use of the term "unit test" in casual conversation or by implication from xUnit tools.

I call them "**unit tests**," but they don't match the **accepted definition** of unit tests very well

Tests as defined in this book don't have any of the characteristics of "unit tests" as described in our earlier definition around **isolation**.

Kent Beck, TDD By Example

Observation

Refactoring is one of the three steps in TDD. If you don't refactor much, it's a smell you are thinking too much upfront.

By this we mean the **contract** that your code exposes to other callers. Your test is an expression of that observable behavior.
TDD is **contract-first**.

Refactoring (noun): a **change made to the internal structure** of software to make it easier to understand and cheaper to modify **without changing its observable behavior**.

The key idea here: you can change your code's details without changing the tests. That is refactoring. It's **safe** because the behavior to preserve is expressed by the test!

https://martinfowler.com/bliki/DefinitionOfRefactoring.html

Observation

In other words, when we change the implementation without changing the **contract** of what is under test, then the tests don't change.

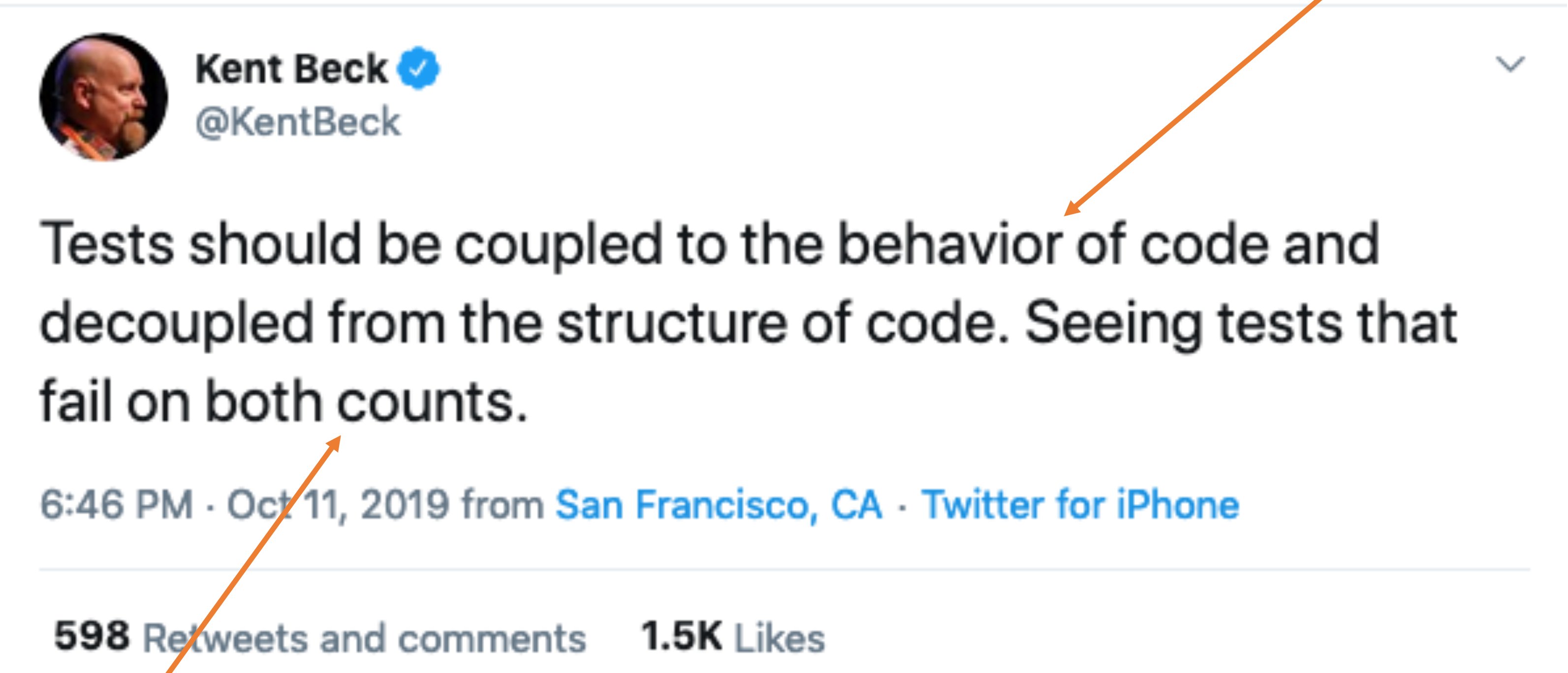
If the program's **behavior is stable** from an observer's perspective, **no tests should change**.

TDD is a **Contract-First** approach to testing. Behavior in this context means that **contract**.

https://medium.com/@kentbeck_7670/programmer-test-principles-d01c064d7934

Observation

Our tests are coupled to the **contract** expressed by the code



Kent Beck
@KentBeck

Tests should be coupled to the behavior of code and decoupled from the structure of code. Seeing tests that fail on both counts.

6:46 PM · Oct 11, 2019 from San Francisco, CA · Twitter for iPhone

598 Retweets and comments 1.5K Likes

Our tests should not couple to the implementation details i.e. via mocks that check details.

https://twitter.com/KentBeck/status/1182714083230904320

Observation

Tests should not use mocks to isolate the SUT, so they are not unit tests!!!!

My personal style is I just **don’t go very far down the mock path**… your test is completely **coupled to the implementation not the interface**… of course you can’t change anything without breaking the tests

The consequence of using **mocks** to **observe the indirect outputs** will be coupling of tests to details…

Kent Beck https://www.youtube.com/watch?v=z9quxZsLcfo

Definitions

The fundamental principle of unit testing.

Failure of a Unit Test shall implicate **one and only one unit**. (A method, class, module, or package.)

Failure of a Programmer (or Developer) Test, under Test Driven Development, **implicates only the most recent edit**.

The fundamental principle of TDD

https://wiki.c2.com/?ProgrammerTest

https://wiki.c2.com/?DeveloperTest

Statement

Use this name, or Programmer Tests, to avoid confusion with unit testing principles.

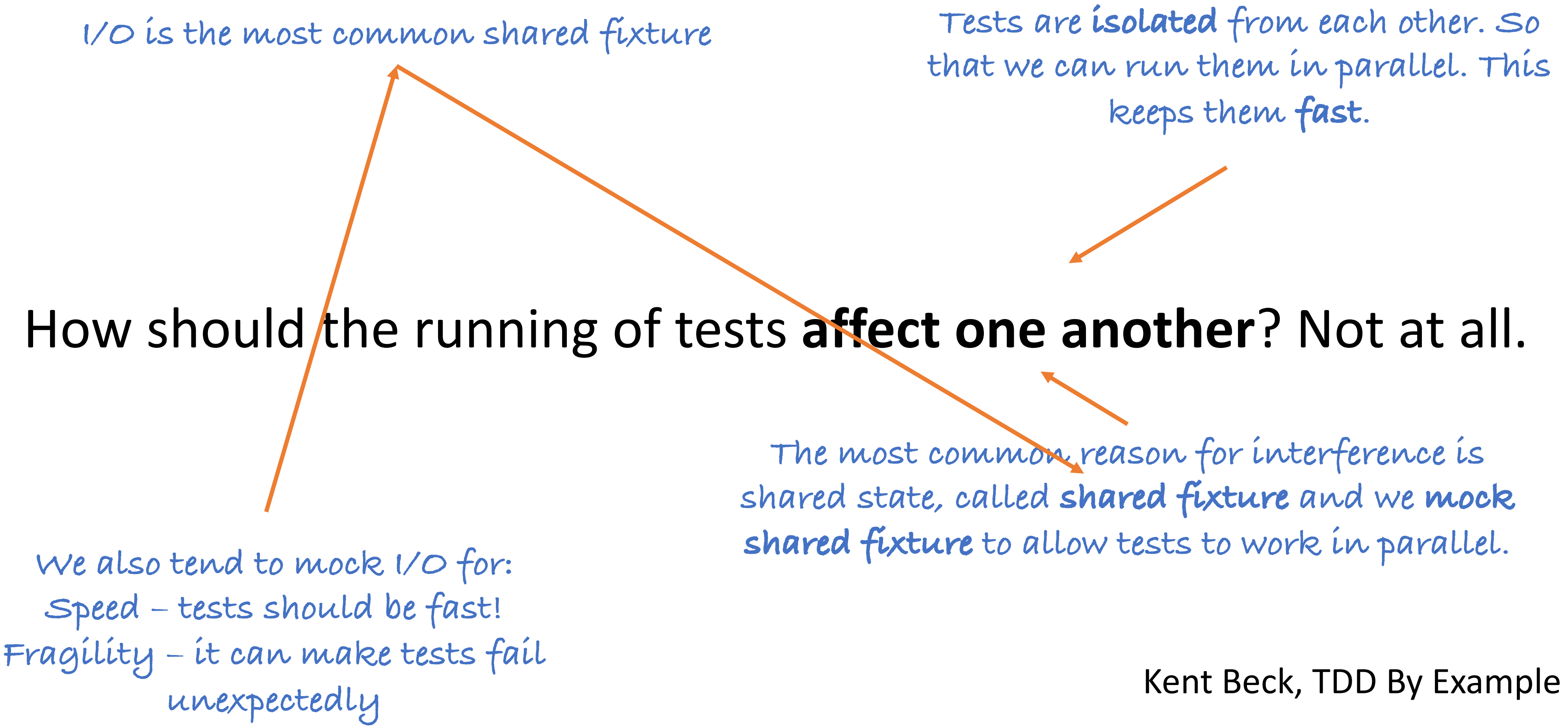
Note that TDD is a process of discovery

Test Driven Development produces Developer Tests. The failure of a test case implicates only the **developer's most recent edit**. This implies that developers **don't need to use Mock Objects** to split all their code up into testable units. And it implies a developer may always avoid debugging by reverting that last edit.

Don't use mocks to isolate the SUT when doing Developer Tests. It is a different practice. Know which practice you are using and its trade-offs.

https://wiki.c2.com/?UnitTest

Observation



How should the running of tests **affect one another**? Not at all.

Kent Beck, TDD By Example

Tests are **isolated** from each other. So that we can run them in parallel. This keeps them **fast**.

The most common reason for interference is shared state, called **shared fixture** and we **mock shared fixture** to allow tests to work in parallel.

I/O is the most common shared fixture

We also tend to mock I/O for:
Speed – tests should be fast!
Fragility – it can make tests fail unexpectedly

Fallacy

2: The trigger for a new test is a new function

Definition

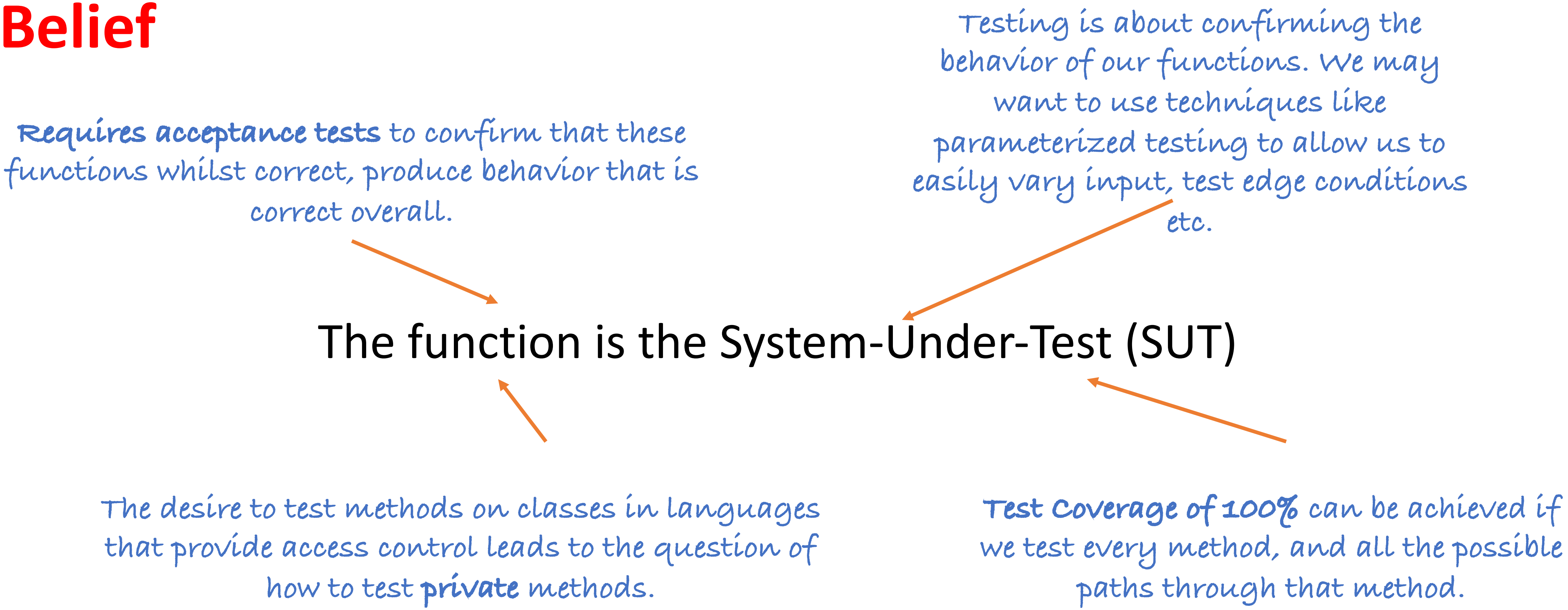
A function has **pre-conditions** and **post-conditions**, a test simply asserts that for a given set of pre-conditions, we get the relevant post-conditions

Write a test that **defines a function** or improvements of a function

Implementation is simply the algorithm to turn the pre-conditions into the post-conditions

https://en.wikipedia.org/wiki/Test-driven_development

Belief



The function is the System-Under-Test (SUT)

Requires acceptance tests to confirm that these functions whilst correct, produce behavior that is correct overall.

Testing is about confirming the behavior of our functions. We may want to use techniques like parameterized testing to allow us to easily vary input, test edge conditions etc.

The desire to test methods on classes in languages that provide access control leads to the question of how to test **private** methods.

Test Coverage of 100% can be achieved if we test every method, and all the possible paths through that method.

Experience

If we returned because it has gone red – it is breaking – why has is broken? Is it because the **acceptance criteria** or our **implementation** changed?

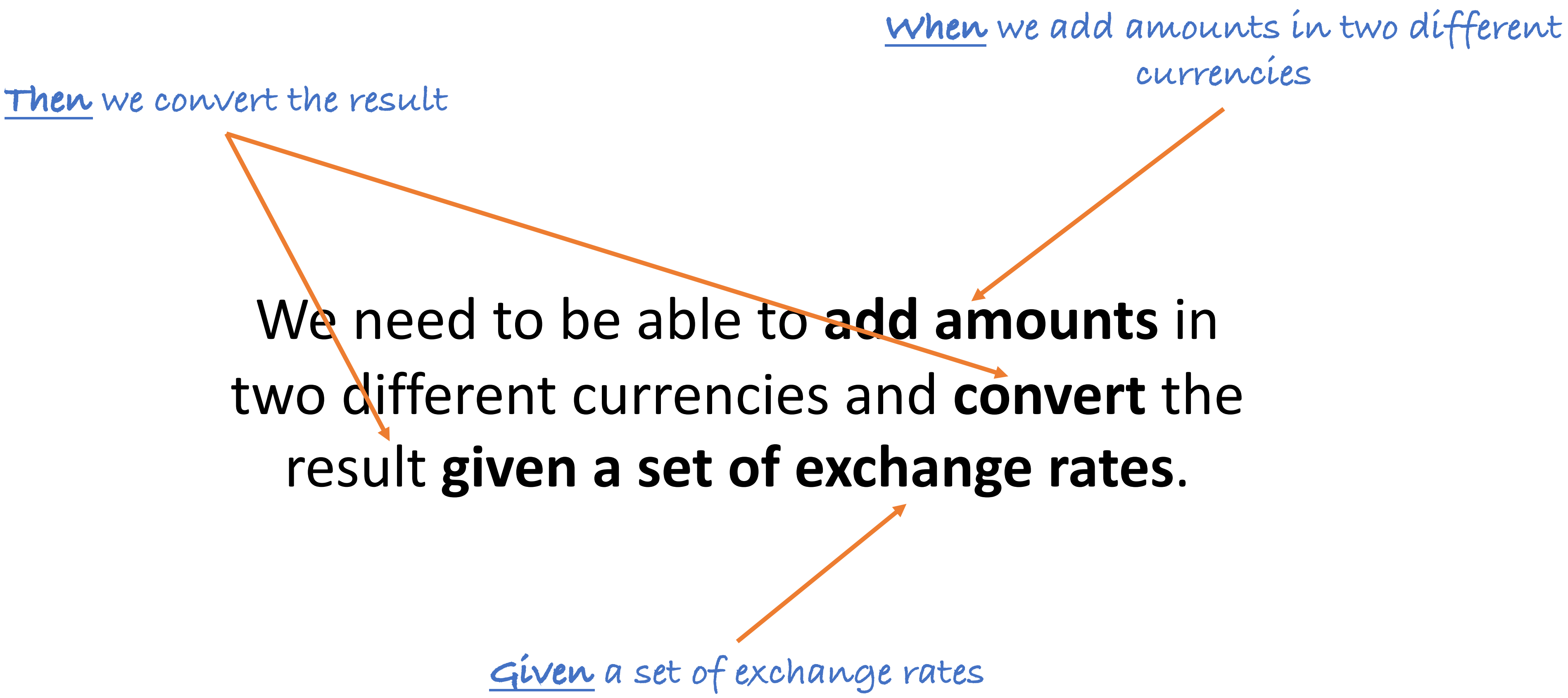
When we return to our tests – it is often difficult to understand their intent

A promise of TDD was **executable specifications**. We would not need documentation, because our tests would document how to use our code through clear examples. Yet in many cases our tests are just **confusing**.

Principle

2: The trigger for a new test is a new behavior

Observation



Then we convert the result

When we add amounts in two different currencies

We need to be able to **add amounts** in two different currencies and **convert** the result **given a set of exchange rates**.

Given a set of exchange rates

Kent Beck, TDD By Example

Observation

Given the state of the world before the test

When we structure our test we can represent GWT as the **Four-Fold Test** (Setup [Given], Exercise [When], Verify[Then], Teardown - **Meszaros**

We can also use **Act, Arrange, Assert** – Bill Wake

Given a set of exchange rates,
When I add two amounts in different currencies together,
Then I get a result in the first currency.

When I exercise the behavior under test

Then we expect the following changes

GWT from BDD by Daniel Terhorst-North

Observation

This switch is really about moving away from understanding TDD as a technique for **testing**, to a contract first technique for **exploring** how an API solves **requirements**

I found the shift from thinking in **tests to thinking in behaviour** so profound that I started to refer to **TDD** as BDD, or **behaviour- driven development**.

At first BDD is just a name for TDD that doesn't carry the confusion around **testing**.

Later BDD becomes a practice, with specification by example tooling, lifecycle. This is **not about that** BDD.

https://dannorth.net/introducing-bdd/

Statement

This is: the **question that answers everything**: what test do I write next?

What is the **smallest change** you could make to the SUT that expresses a change to the acceptance criteria for a behavior? **Test that.**

The **next test** you write in TDD is just the **most obvious step** that you can make towards implementing the **requirement** given by **a use case or user story**.

Wait, **if TDD captures requirements, what are Acceptance tests for?** More on this later.

A use case or user story tells us what a customer needs us to build – the behavior that the system should exhibit. The acceptance criteria for that drive our tests.

Observation

By implication this must not be exported – public – but be hidden – private – as it can't be part of the contract. So it is **already covered by the existing tests.**

Remember that **green phase** is a **transaction script** – discovering the algorithm – so we have **poor structure,** that emerges in refactoring

You do not **write new tests** if you **introduce new methods when refactoring** to clean code.

This could be a new **class** too As long as it is a detail of refactoring.

Refactoring is changing the implementation without changing the behavior – **we do not change the contract when refactoring**

Fallacy

3: Customers write Acceptance Tests

Definition

This requires us to author a tool that supports Data-Driven Tests like Fit or DSL scripting like Cucumber;

We test the story not a unit – but isn't that TDD?

Originally called Functional Tests because each acceptance test tries to test the functionality of a **user story**.

Acceptance tests are different [is] **modeled** and **possibly even written** by the **customer**. ...Hence the even-newer name, Customer Test.

If the **Customer defines the acceptance criteria**, can they write a **test that expresses this**? A script that exercises the software?

On-site customer was an important XP concept – a domain expert the team could question, often replaced with a Product Owner today

https://wiki.c2.com/?AcceptanceTest

Experience

Remember these are **Customer** Tests, that is why we use FIT or Cucumber et al. to facilitate their interaction i.e. **writing the tests**.

These two problems--that **customers don't participate**, which eliminates the purpose of acceptance testing, and that they **create a significant maintenance burden**, means that acceptance testing isn't worth the cost. I no longer use it or recommend it.

This is key: acceptance tests written using FIT or Cucumber are more expensive to write, because you need to translate to inputs, and more **expensive to own** as they **expensive to change**

James Shore, http://www.jamesshore.com/Blog/The-Problems-With-Acceptance-Testing.html

Helped write **FIT**. So he is not just a critic, he **built the tooling** we are talking about.

Experience

ATDD only exists because we don't believe that TDD does this.

ATDD is perilous because it implies that TDD does not deal with the **acceptance criteria** for user stories

*Remember we established earlier that **TDD is driven by acceptance criteria** from a user story. So there is **no difference in intent**.*

Experience

Another aspect of ATDD is the **length of the cycle** between test and feedback. If a customer wrote a test and ten days later it finally worked, **you would be staring at a red bar** most of the time.

Kent Beck, TDD By Example

If the tests are nearly always red, developers don't run the ATDD suite until the end. And if they are integrating with others, they miss integration issues and have to scramble to get the tests passing.

Principle

3: Customers specify Acceptance Criteria

Statement

Example-driven development is another name for this style – but its really TDD done right. This produces the GWT we use for our test

Customers illustrate their descriptions with concrete examples…programmers use these examples to guide their work…Sometimes [programmers] use the examples directly in their tests…More often…programmers use the **examples as a guide**, writing a multitude of **more focused, programmer-centric tests** as they use TDD

This also achieves self-documenting code

https://www.jamesshore.com/v2/blog/2010/alternatives-to-acceptance-testing

Fallacy

4: It doesn’t matter if you are test first or test last

Definition

Test last does tests **after** software is written. It is conventional unit, integration and acceptance testing, but practiced by developers with xUnit tools.

A development process that entails executing unit tests **after** the **development** of the corresponding units is **finished**.

Implicitly **modeling occurs before development**. This may be a lightweight process like CRC cards, or heavier exploration via UML

The **feedback loop is long**. The design takes to implement. In a RAD environment this may be a few days, it might be following iteration though or beyond.

Experience

How do we know when we are Done?

If you write code that is **not needed by the given requirements** you are engaging in speculation. Most likely you will be wrong. The code will not be **needed** or you will have to **re-work** that code.

Whilst we may think it will be needed, often the cost-value turns out to be poor and the **customer doesn't want it**. But we already paid for it.

Even if we guess right, most likely we have to re-work because the requirements are not right. In the worse case we refuse to abandon our speculation and force it to work with **hacks**.

Principle

4: Only write production code in response to a test

Statement

We can't write speculative code here - only code that has a requirement.

If you don't have acceptance criteria or a clear requirement, it's a prompt for a conversation with the Customer – only build it once **'Done'** is defined.

Only write **production code** in **response to a test**.
Only write a **test in response to a requirement** (user story & acceptance criteria).

Don't forget, this tells us what the contract we are defining should do – its behavior.

Observation

Test First is **design-by-contract**. We are guided by the behavior required of the system.

You need a way to think about **design**, you need a method for **scope control**

If we test first we don't end up with speculative code. We know when we are done, and our code is a simple as it needs to be, but no simpler.

Kent Beck, TDD By Example

5: You want 100% test coverage of your code

Definition

If we cannot write code without a test because of TDD, then all of our code MUST be covered by tests.

TDD followed religiously should result in **100 percent** statement **coverage**

We only get a discrepancy if:
(a) We have speculative code, not needed by a test
(b) We introduce an untested branch during refactoring

Kent Beck, TDD By Example

Experience

Although the team is practicing TDD, not all the code may be exercised by TDD. That may lower our coverage.

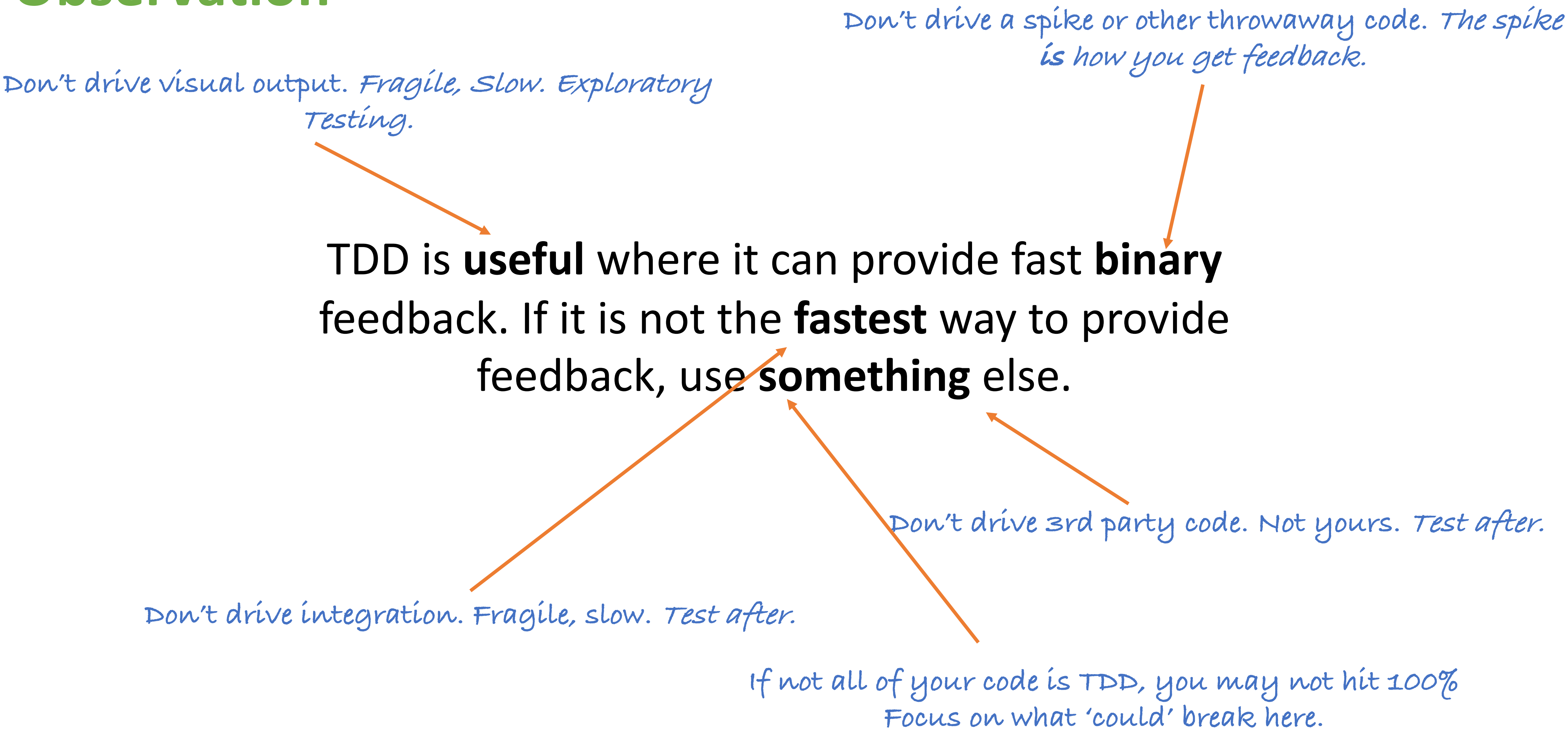
Many test suites where development teams **practice TDD** have less than **100%** test code.

Is the amount of coverage important when we refactor, or is it a **lowering of test coverage** that matters?

Principle

5: Not all of code should be driven by TDD

Observation



TDD is **useful** where it can provide fast **binary** feedback. If it is not the **fastest** way to provide feedback, use **something** else.

Don't drive visual output. *Fragile, Slow. Exploratory Testing.*

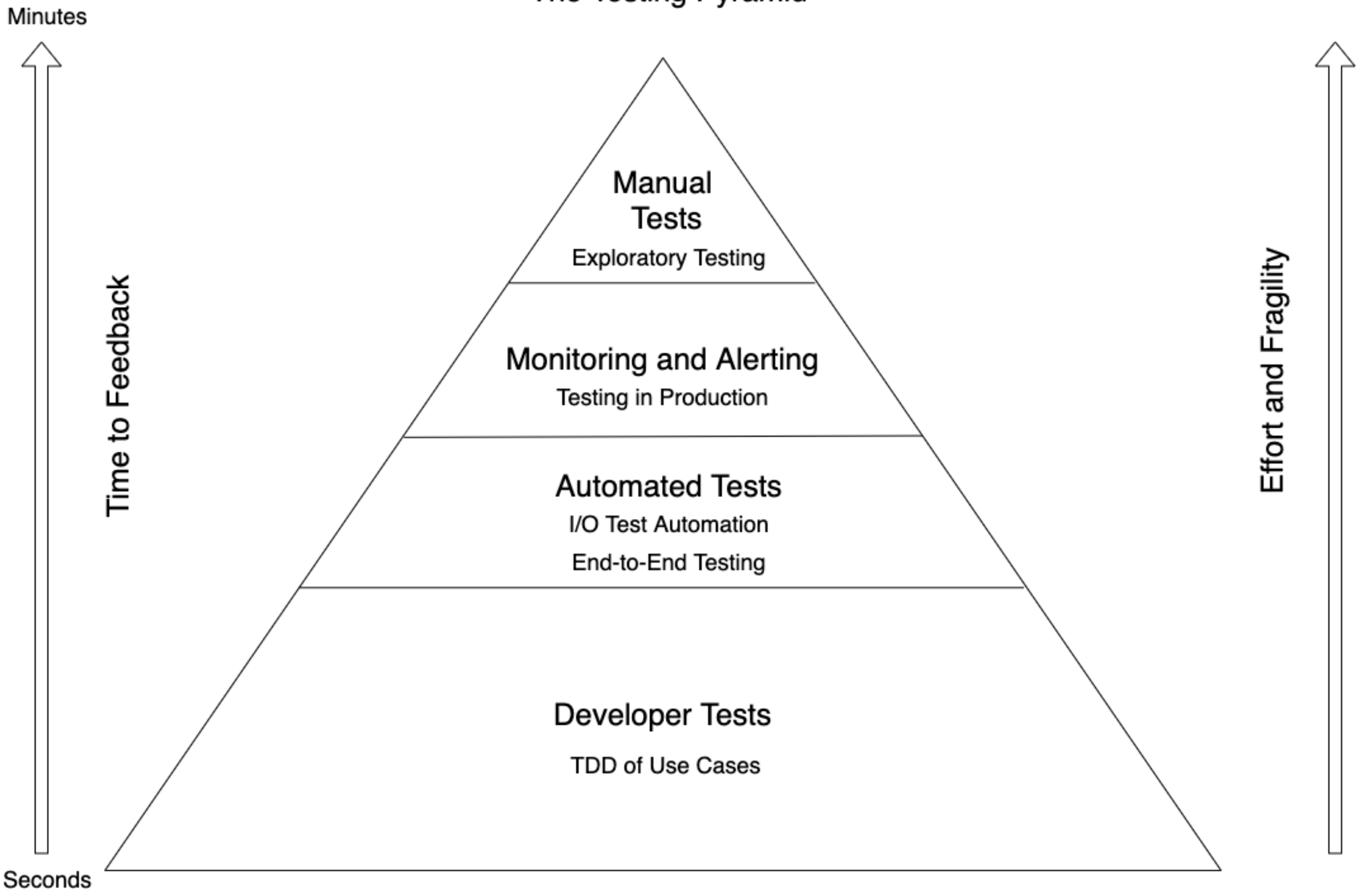
Don't drive a spike or other throwaway code. *The spike **is** how you get feedback.*

Don't drive 3rd party code. Not yours. *Test after.*

Don't drive integration. Fragile, slow. *Test after.*

If not all of your code is TDD, you may not hit 100%
Focus on what 'could' break here.

The Testing Pyramid



Minutes

Time to Feedback

Seconds

Manual Tests

Exploratory Testing

Monitoring and Alerting

Testing in Production

Automated Tests

I/O Test Automation

End-to-End Testing

Developer Tests

TDD of Use Cases

Effort and Fragility

Fallacies & Principles

1: Developers write Unit Tests

2: The trigger for a new test is a new function

3: Customers write Acceptance Tests

4: It doesn’t matter if you are test first or test last

5: You want 100% test coverage of your code

1: Developers write Developer Tests

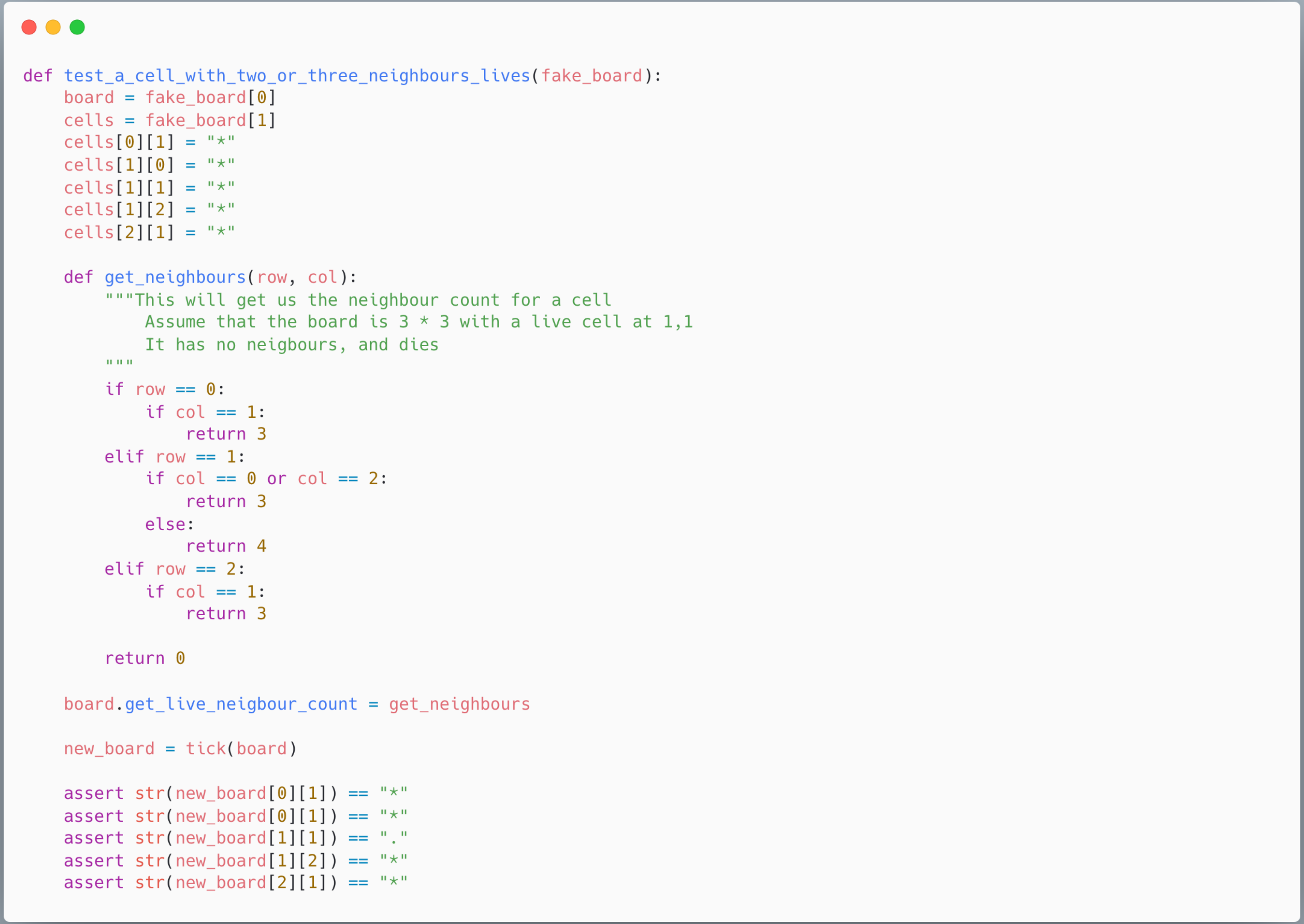
3: The trigger for a new test is a new behavior

2: Customers write Acceptance Criteria

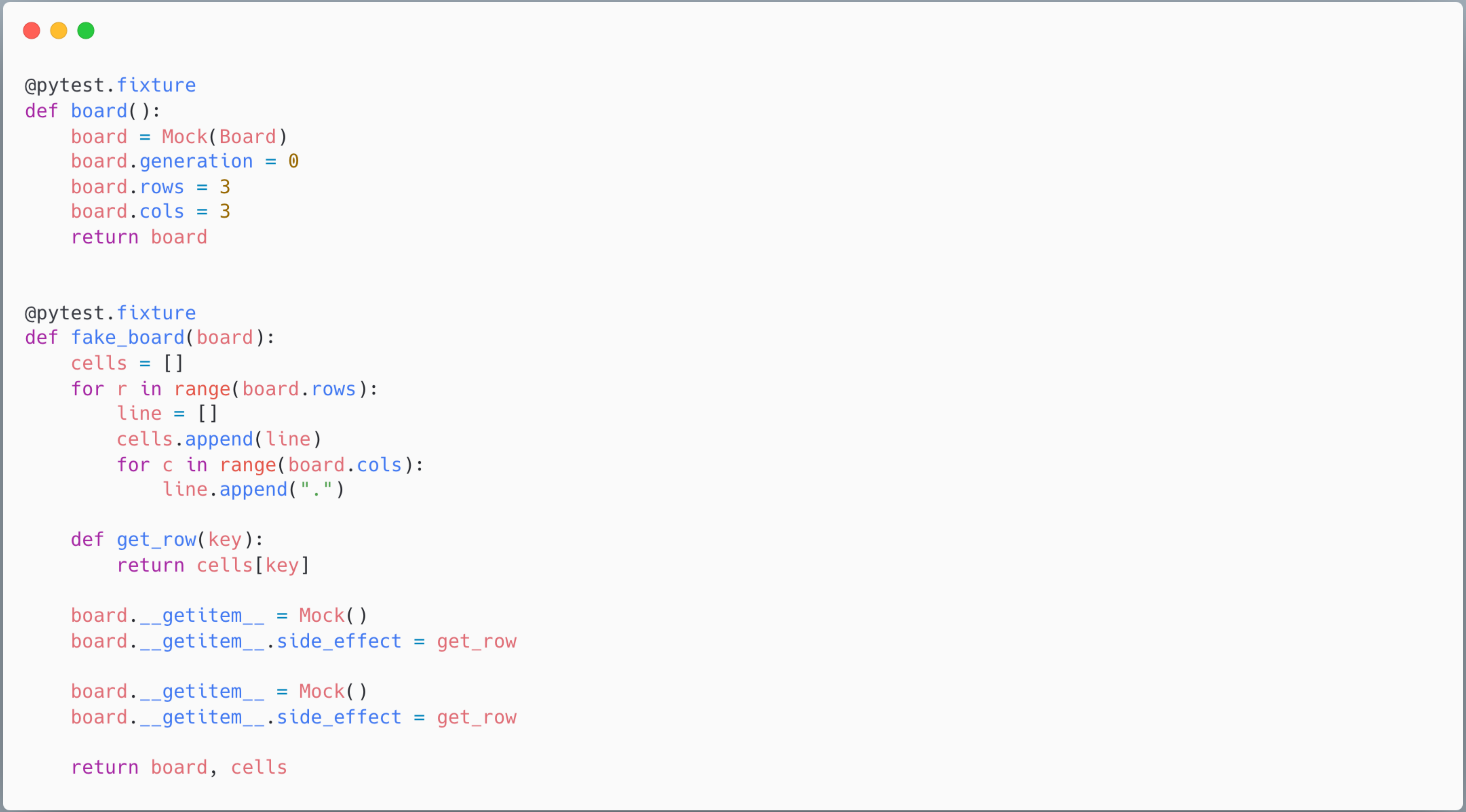
4: Only write production code in response to a test

5: Not all of code should be driven by TDD

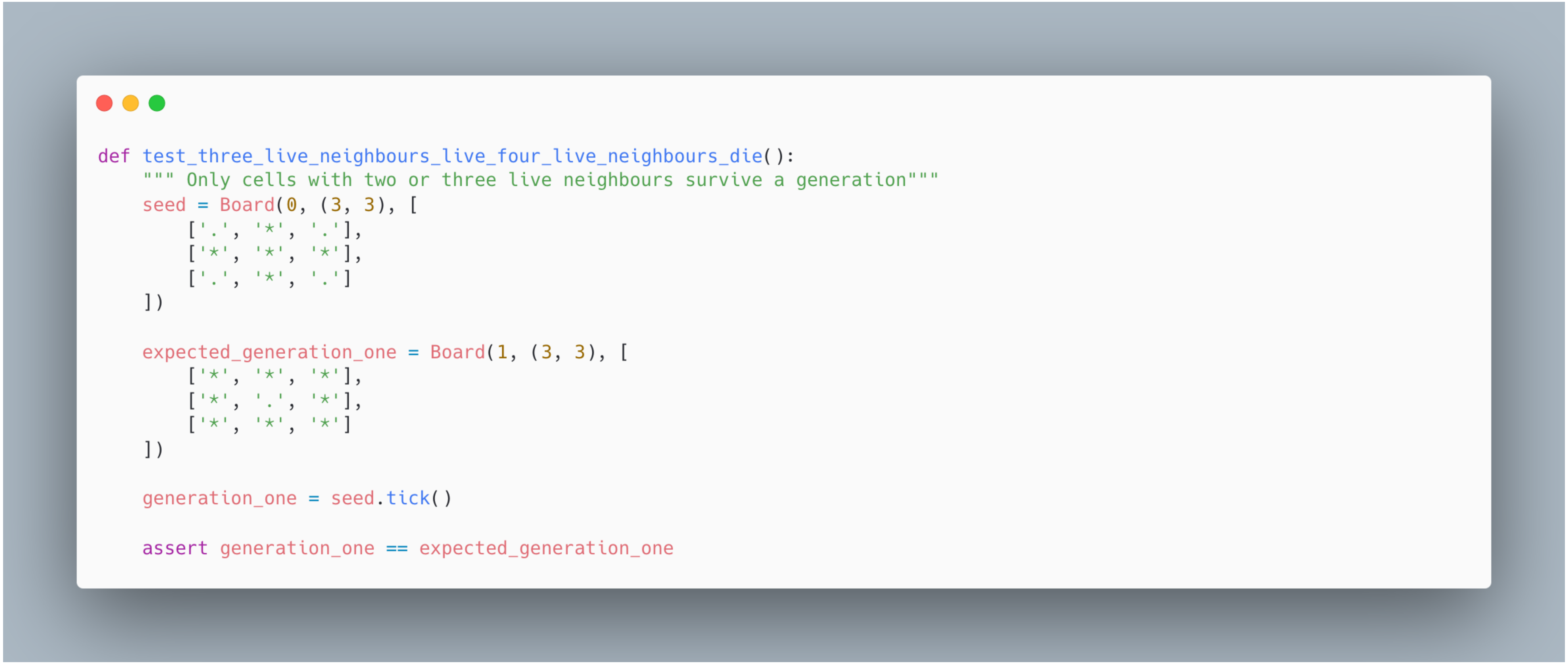
Examples



```python
def test_a_cell_with_two_or_three_neighbours_lives(fake_board):
    board = fake_board[0]
    cells = fake_board[1]
    cells[0][1] = "*"
    cells[1][0] = "*"
    cells[1][1] = "*"
    cells[1][2] = "*"
    cells[2][1] = "*"

    def get_neighbours(row, col):
        """This will get us the neighbour count for a cell
            Assume that the board is 3 * 3 with a live cell at 1,1
            It has no neigbours, and dies
        """
        if row == 0:
            if col == 1:
                return 3
        elif row == 1:
            if col == 0 or col == 2:
                return 3
            else:
                return 4
        elif row == 2:
            if col == 1:
                return 3

        return 0

    board.get_live_neigbour_count = get_neighbours

    new_board = tick(board)

    assert str(new_board[0][1]) == "*"
    assert str(new_board[0][1]) == "*"
    assert str(new_board[1][1]) == "."
    assert str(new_board[1][2]) == "*"
    assert str(new_board[2][1]) == "*"
```



```python
@pytest.fixture
def board():
    board = Mock(Board)
    board.generation = 0
    board.rows = 3
    board.cols = 3
    return board


@pytest.fixture
def fake_board(board):
    cells = []
    for r in range(board.rows):
        line = []
        cells.append(line)
        for c in range(board.cols):
            line.append(".")

    def get_row(key):
        return cells[key]

    board.__getitem__ = Mock()
    board.__getitem__.side_effect = get_row

    board.__getitem__ = Mock()
    board.__getitem__.side_effect = get_row

    return board, cells
```



```python
def test_three_live_neighbours_live_four_live_neighbours_die():
    """ Only cells with two or three live neighbours survive a generation"""
    seed = Board(0, (3, 3), [
        ['.', '*', '.'],
        ['*', '*', '*'],
        ['.', '*', '.']
    ])

    expected_generation_one = Board(1, (3, 3), [
        ['*', '*', '*'],
        ['*', '.', '*'],
        ['*', '*', '*']
    ])

    generation_one = seed.tick()

    assert generation_one == expected_generation_one
```

End