First Module - Overview

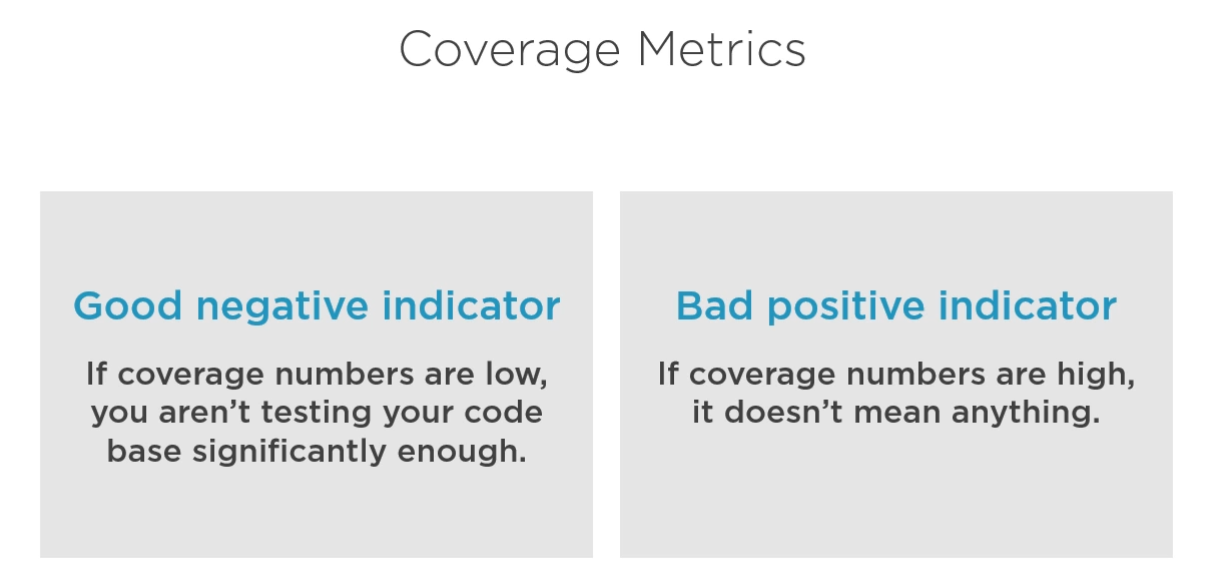
Unit Test

* Verifies some small piece of code
* Runs quickly
* Works in isolation

Benefits:

* Confidence
  + Means that you can make changes of your code base that they will not break existing functionality
* Move with a faster pace
* Maintain amount of technical debt

\*\*Your unit tests **must** have appropriate assertions



\*\*Spend most of the time on testing business logic

**Pragmatic Approach:**

Carefully choose code to test => Use the most valuable tests only

Writing code is an expensive way to solve problems. The more code you have, the more you extend the surface area for potential bugs.

**What is a valuable test?**

**Has a high change of catching a regression error**

Is related the amount of code that the test traversed during the execution,

The more code is executed, the higher chance there is, that this test be able to reveal an error

Not only the number of lines that matter, but also the significant of those lines!

The code that represents the business logic is certainly more important than some utility code

It’s not only your code that counts here, but also the code you didn’t write (libraries and frameworks etc.)

**Has a low chance of producing a false positive**

False positive = false alarm, a result indicating that a test failed, while in the reality the functionality it covers hasn’t been broken

Usually take place during the refactoring process, when you modify the implementation without changing the actual functionality

False positives can have a devastating effect on the health of test suit, they dilute the ability to quickly spot the problem in case of something goes wrong

The number of false positives you get is directly related to the way that test verifies the correctness of SUT.

The more that test is tight to implementation details 🡺 the more false alarms

The only way to reduce false alarms 🡺 Decouple tests from implementation details as much as possible

**We should make sure to verify the end result that your code generates, NOT the actual steps it takes to do that**

**Provides fast feedback**

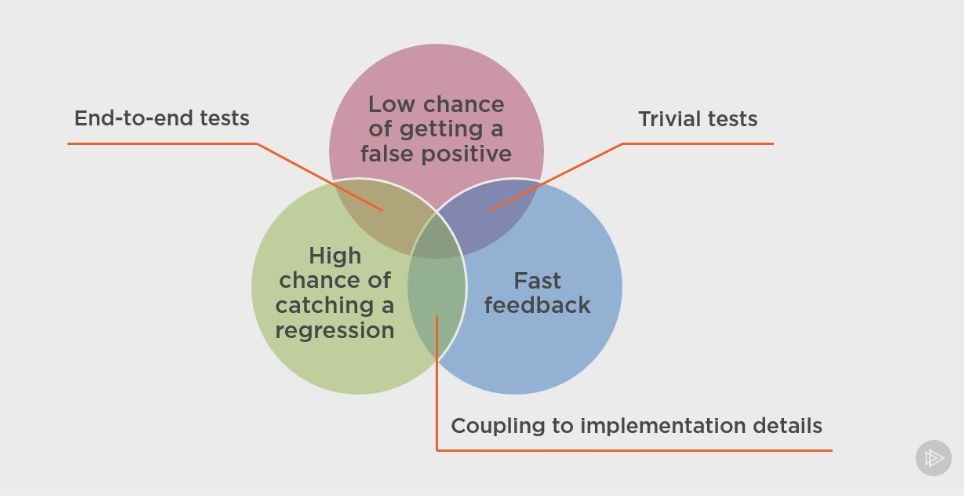
The shorter the feedback loop, the faster you can adjust your course and less effort you waste going in a wrong direction

Can only be provided by fast test suit

**Has low maintenance cost**

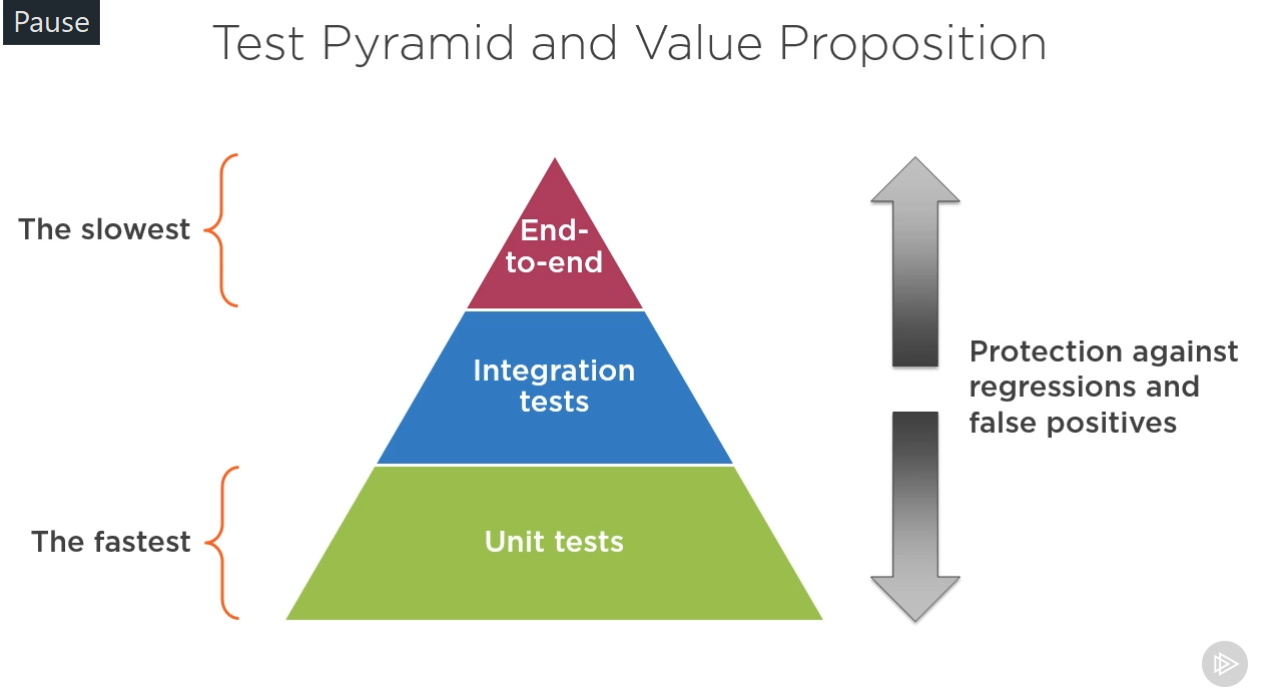
A product or the number of lines you used to write a test

The fewer that number 🡺 more maintainable test becomes

****

The first three are mutually dependent, it is impossible to maximize one of them without damaging the others.

Key to building a great test suit 🡺 Finding a balance between the first 3 components



Second Module - Introduction

Styles of unit testing:

1. Output verification 2- State verification 3- Collaboration verification

**Hexagonal Architecture (Ports and Adapters):**

Domain model is the most important part of the application

* Domain model doesn’t communicate with the outside world
* Shouldn’t know how it persisted to the database
* Responsible for holding business logics

Application services handle all communications

* Don’t contain business logic
* Translator between domain and outside world

Ports are the public APIs through which external systems communicate with your application

They are basically the facet of your hexagon

Adapters are the external systems themselves; they adapt or connect your system through the publicly available ports

\*\*The number of ports any systems can have, is arbitrary, and it shouldn’t be 6 because of hexagon!

**What is an implementation detail?**

Public API is any members of class or application layer (as a whole) that can be used outside of that class or application layer (basically any members that mark with *public* keyword)

A well-crafted API shouldn’t have any implementation details exposed.

Public API ≠ Implementation detail

**How to determine if a class member is an implementation detail?**

Address an immediate goal of the client code

Addresses that goal completely

Number of operations to achieve a single goal > 1?

Exposing implementation detail: Well-defined API

Usage from the outside layer counts, **NOT** from the same layer

Some language contracts such as interfaces or public/private keywords are **irrelevant** when identifying an implementation detail 🡺 if that interface doesn’t help to solve a particular problem the client code has, or does it partially, that means it also exposes implementation details to the outside layer, similarly if you make some method public, it doesn’t automatically make a well-defined API, it may ever be an implementation detail.

If a method or a class doesn’t address some immediate requirements from the outside world, it is a good idea to hide it

-------------------------------------------------------------------------------------------------------------------------------

Third Module – Styles of Unit Testing

Output verification or Functional Style:

* Best protection against false positives
* Easy to maintain
* Only suitable for functional code

State Verification:

* Good protection against false positives
* Should verify through the public API

\*\* Don’t extend the public API just to satisfy tests, it’s a code smell

As long as the SUT encapsulation is not violated, state verification is a good approximation to the functional style of unit testing

* Reasonable maintenance cost

Collaboration Verification:

Communication between domain classes in hexagonal are implementation details.

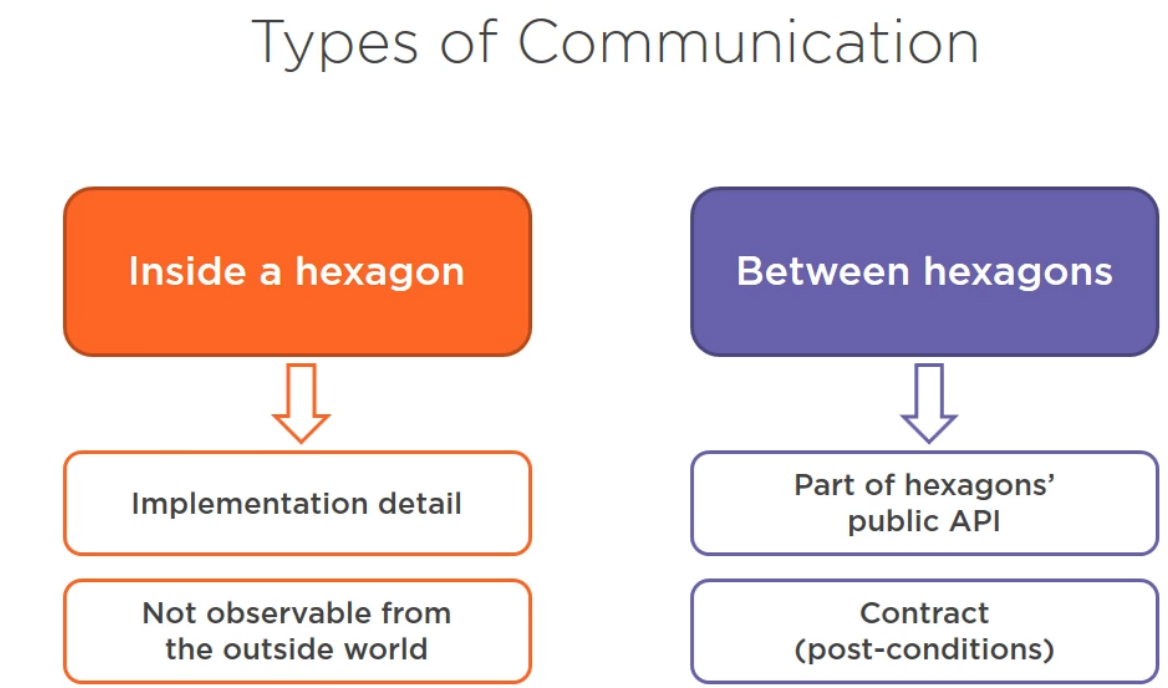
The collaborations your domain classes go through to achieve some goals are not part of public API, therefore binding unit test to the way those classes communicate with each other, introduces coupling between the SUTs and implementation details.

Such coupling makes the test produces a lot of false positives, as the collaboration pattern tends to change often during the refactoring

* Lots of false positives
* Maintainability is worse
* Verify the communication pattern between your application and external systems such as Message Bus or 3rd party systems

\*\*Always wrap your services with your own gateways. You should mock only types that you own.

\*\*Communication with other hexagons should be in application services



Result of comparing unit test styles:

* Adhere the functional style the most
* State verification is the second-best choice (Only verify through public APIs not implementation details)
* Collaboration verification is good for communications between applications

And bad for communications inside the domain model

Mocking internals of your domain model is a design smell, try to avoid this as much as possible

Black box testing 🡺 Testing without knowing the internal structure

White box testing 🡺 Testing internal structure

\*\*Try to adhere black box testing at each level as much as possible.

it will help you avoid coupling your test to SUT’s implementation details and will urge you to find ways to verify the observable states of the system instead.

Focusing on the behavior the clients of your code care about, generally result in few false positives.

When writing a unit test, you need to know something about the SUT, however try to make sure that there is as little internal structure leaking to the test as possible.

Does the test verify a business requirement?

If the answer is no, try to find a way to refactor it so that it does, or removing it all together.

The most valuable tests are always tests that have at least some connections to the business requirement your code base is ought to address.

\*\* **BDD** enforces you to write test in a way that describes business requirements for your application without diving into that details of how those requirements should be addressed.

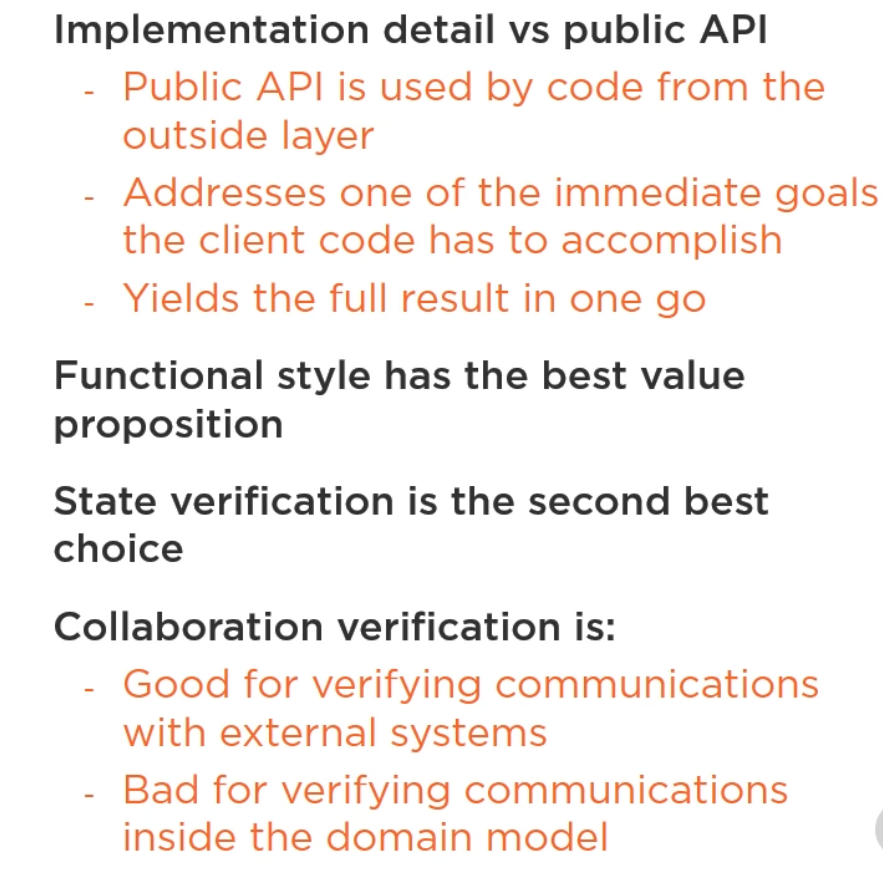
\*\* View your code from the end user’s perspective and write tests to verify its observable behavior **NOT** its internal structures.

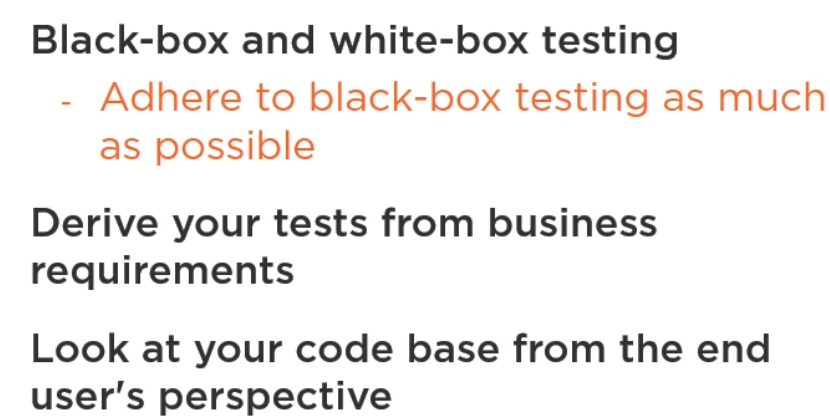
Ask yourself this question, if this test fails, what business requirements is compromised

Tests that have the most business value are those are directly derived from business requirements, the closer you can get this kind of verification is better.

**BDD is TDD done right**

**Summary:**





Fourth Module – Getting the Most Out of Unit Tests

Tests coupled to implementation details 🡺

* Fail once those details are changed
* Produce lots of false positives
* Hard to maintain during refactoring

**Header interfaces:**

Header interfaces are the interfaces that have a single implementation only which fully mimics that interface.

Header interfaces don’t represent abstractions and generally considered a code smell.

Collaboration verification inside the domain model 🡺 Header interfaces

To employ this type of units testing, you need to substitute SUT’s neighbors with mocks. And to do that introduces a separate interface for each such a neighbor.

As a result, you end up an overcomplicated solution, because for each concept of your domain model, you need to bring in two types of class with the actual implementation and the interface that fully mimics the class.

**Cycling dependencies:**

Cycling dependencies are dependencies that well-cycled and don’t have clear beginning because of that.

They add a tremendous cognitive load when you try to read and understand the code. With such dependencies **you don’t know where to start** from to understand what one of the classes does, you need to push the whole graph siblings into your head.

**TRY TO AVOID CYCLING DEPENDENCIES AND USE TREE-LIKE CLASS STRUCTURE**

**Domain Isolation:**

Domain model should be self-contained, meaning that classes in it should not talk to outside of the model boundary.

WHY DOMAIN ISOLATION?

* Such isolation brings a clear separation of the 2 concerns:

1 – Holding the domain knowledge

2 – Communicating with the outside world

Such separation in turn allows to reduce the cognitive load when you reason about the code base as you can focus on single thing at a time.

On the contrary, conflating these two notions often leads to overcomplicated solutions

* Testability

Having a self-contained domain model with classes that don’t refer to external dependencies, allows you to test it using the first two style of unit testing (output, state), and that means you can drastically increase the value of your test suit as you can avoid the use of mocks when testing the domain logic.

This pattern when you remove all external dependencies from code with some important business logic to unit tested is called Humble-Object.

\*\*The more important or complex the code is, the fewer external dependencies it should have.

You need to separate the code that communicate with the external world from the code that does the hard lifting.

**Types of code and external dependencies:**

1. **Domain model, algorithms**:

They carry a complex or important logic in them, but the same time don’t have lots of external dependencies. Ideally, they shouldn’t have any.

1. **Trivial code**:

The code that doesn’t do much and at the same time doesn’t have external dependencies.

1. **Controllers, coordinators**:

They don’t do any critical business work themselves, but coordinate the work of other components

1. **Mess**:

The code that contains both external dependencies and has some complex business-related responsibilities.

Like fat controllers. Controllers that don’t delegate the actual work anywhere and do it themselves.

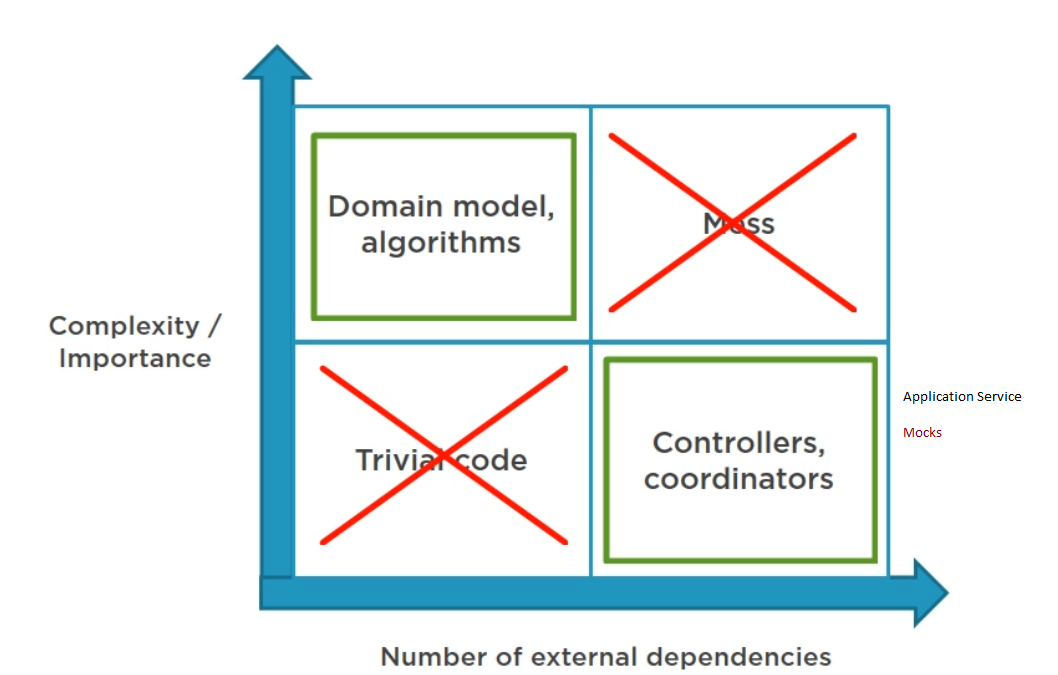
The most problem unit testing is brought the code from Mess.

This code is usually messy and hard to maintain and so are the unit tests that covered it.

The guideline of separating code with external dependencies from the code that does complex work is about moving as much code as possible from Mess into Domain model or Controllers quadrants.

For example, if you have a fat controller you need to extract the algorithmic part of it into the domain class.

The mess quadrant is hard to unit testing because of large number of dependencies and it also scary to leave it untested as it does an important work.



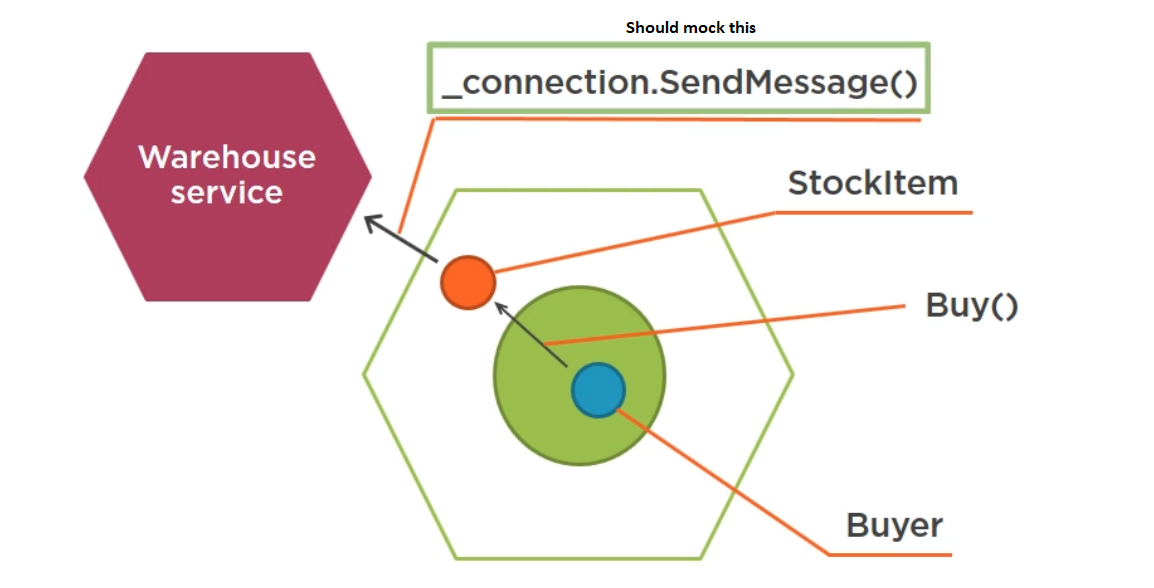
Fifth Module

Sometimes to simplifies the code base, it makes sense not to adhere the CQS principle. Like **stack.Pop();**

Sixth Module – Integration Tests

Verify collaborations at the very edges of your system.

The farther away we keep mocks from the edges of our application, the more we expose ourselves to potential false positives.

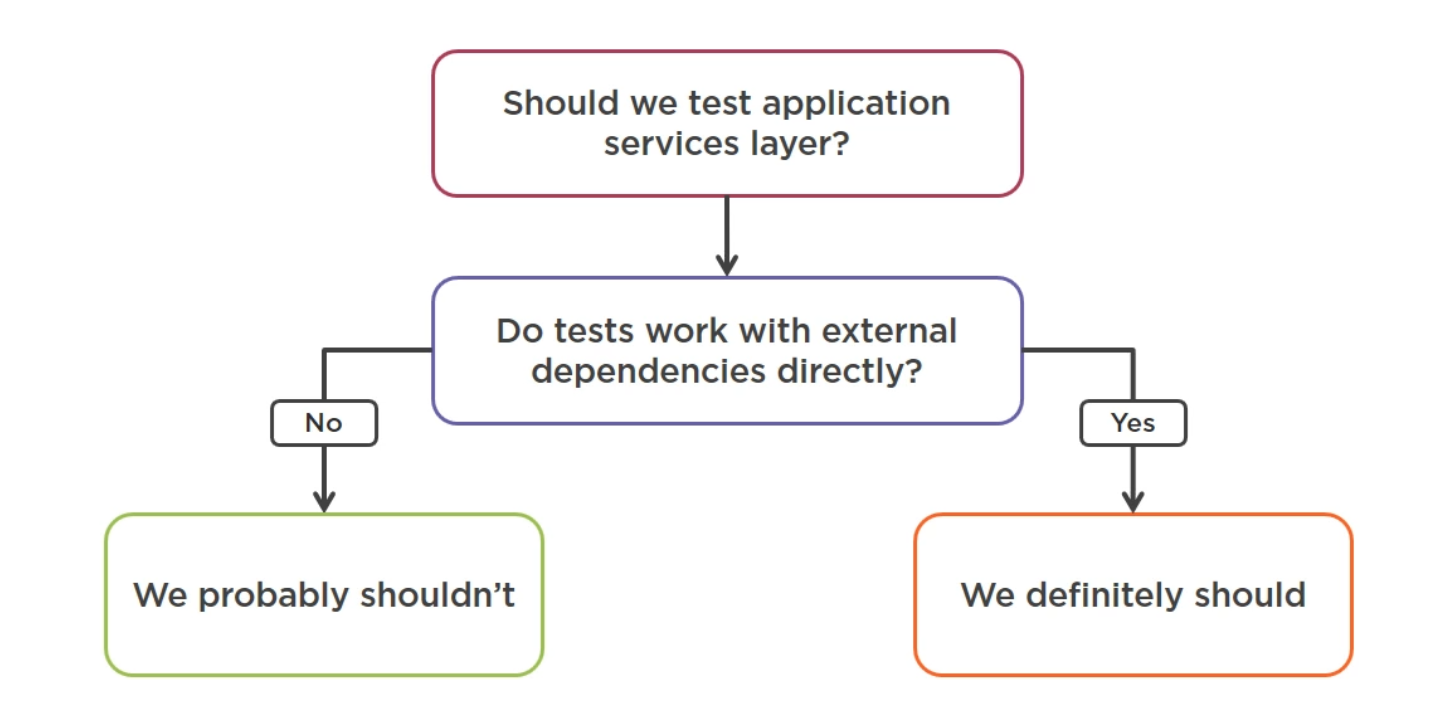


Database is a new hexagon and communication with it is performed by the application service layer.

The goal we are trying to achieve with integration tests, is not to check how domain model itself works, but validate how it combines with other applications.

Types of external dependencies:

* Controllable 🡺 tests against them directly e.g., database, file system, stable uncontrollable dependency
* Uncontrollable 🡺 substitute with test doubles



How to isolate integration tests from each other?

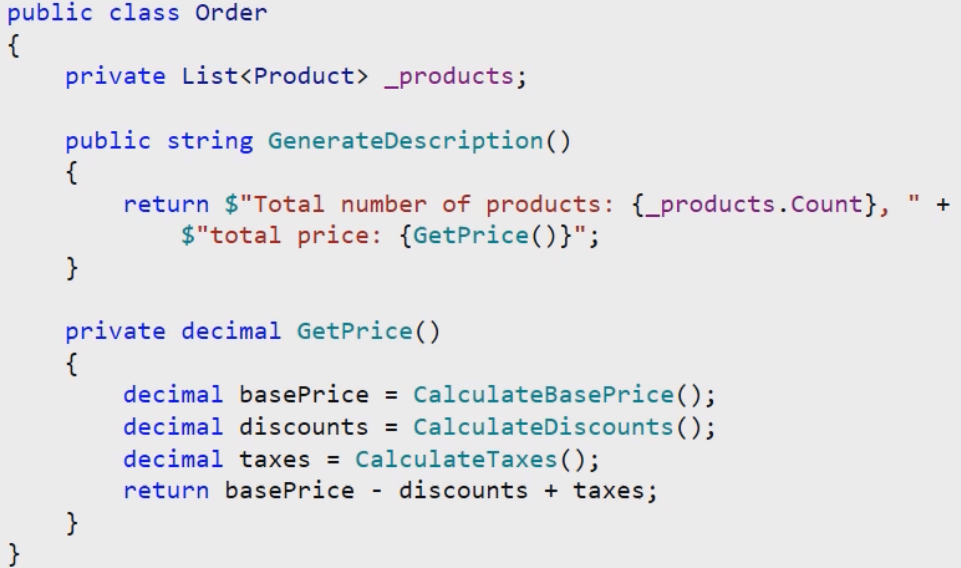
* Run them sequentially
* Remove data left after test execution
  + Restore database backup before each test 🡺 Slow
  + Wipe out data after test execution 🡺 Some data might remain
  + Wrap each test with a transaction 🡺 Can interfere with the SUT’s flow
  + Wipe out all data **before** test execution 🡺 Solve the data clean up problem
    - Each test creates its own set of test data (one test should not depend on what’s left from another one)
    - Only master data should be deleted (data that your application can change)
    - Setup proper database delivery

As a general guideline use unit tests to cover your domain model in isolation from external dependencies and use integration tests to check how the application service layer works with external applications.

Seventh Module – Unit tests Anti-Patterns

1. Exposing Implementation Details

**How to unit test private methods?** You shouldn’t do that. Private methods are an implementation detail not a public API. If you see a private method is too complex and leave it untested is too dangerous, that’s an indicator that you need to extract its inner working into a separate class and tests that class instead.



In the above example we can tested the PriceCalculator class separately from the Order class.

In many cases the domain model itself can also be represented using multiple sub-layers where classes from inner layers comprise a Public-API for classes from the upper ones. In this example PriceCalculator is a small domain service which resides lower than Order in this multi-layer domain model structure and thus it should also comply with the same principles in terms of hiding the implementation details and exposing a well-defined Public-API.

Here this Calculate method does obey this principle and addresses an immediate goal that class needs to achieve that class needs to achieve.

So, while PriceCalculator is not part of domain model Public-API, it does expose a well-defined Public-API for one the layers inside that domain model and thus also needs to be covered by unit tests.

2.Exposing State Getters

We shouldn’t make a class member public just for testing its state, on the contrary we should make our tests work with this class the same way as its clients.

**\*\*Test the observable behavior only**

****

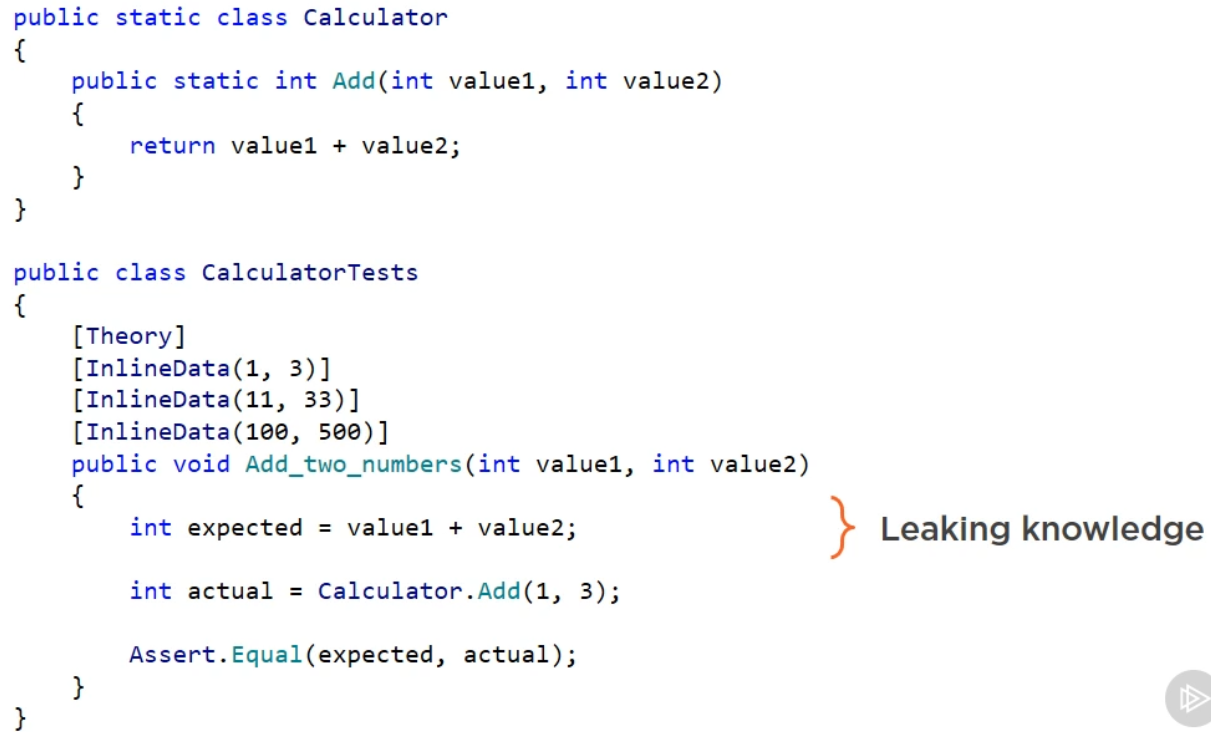
In the above example we must test *Discount* value to verify our customer status’s state

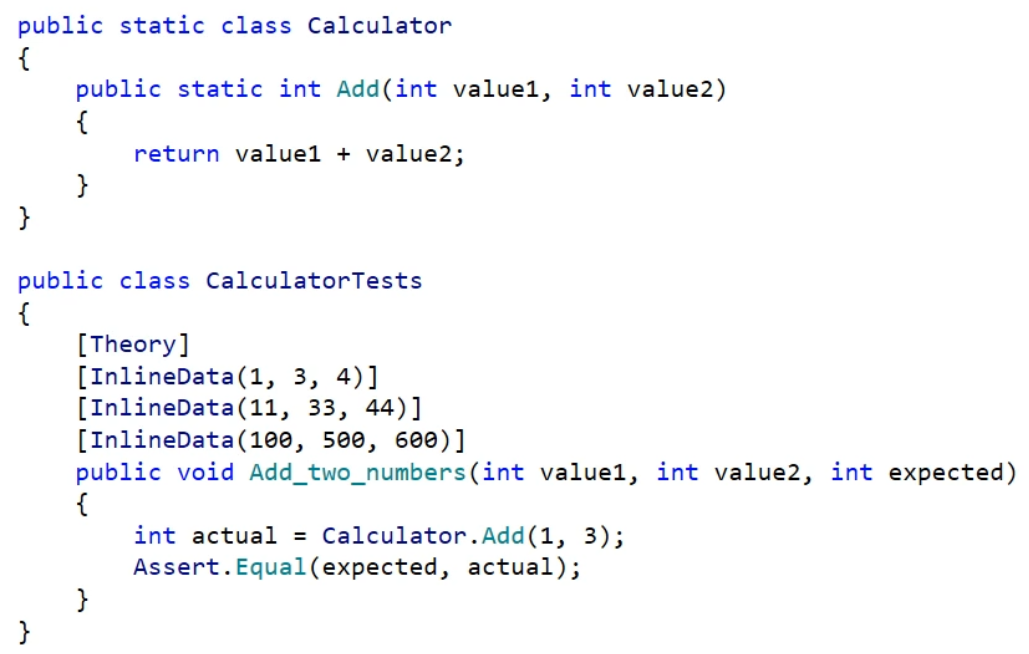
Additional state getters are also often introduced when verifying collaborations between classes. To check the fact that one class calling another, a programmer could make some piece of information on the receiving part public, however this is also code smell as this piece of information wouldn’t have been made public otherwise. We should treat the whole set of collaborating classes as a single SUT and verify the result through the natural APIs.

Exposing the implementation details is almost always a bad idea, if you find yourself making private methods or fields public solely for satisfying a test you need to reconsider that and try to bind it to existing Public-APIs instead.

3.Leaking Domain Knowledge to Tests

We must not reproduce the domain’s algorithms in tests, and we must pre-calculate the results and hard-coded them into tests instead.





4.Code Pollution

Introducing additional code to your main code base just to enable unit testing.

Don’t introduce code to your main code base solely for testing purposes.

* Adding methods
* Adding switchers to the existing methods

5.Overriding Methods in Classes-dependencies

This practice is usually a sign of SRP violation, instead of overriding the single method in the dependency, try to extract a new class out of it and substitute that class with a test double instead.

The fact that you need a substitute not an entire class but only a part of it is a strong sign of SRP violation.

6.Non-determinisim in Tests

It basically stands for Flickering tests. Tests that pass most of the time but fail once in a while and then if you try to run them one more time turn green again.

Non-determinism can have a devastating effect on your entire test suit, once you get a custom to such variable failures, legit tests failures get ignore with them and the test suit no longer perceived as a reliable source of feedback.

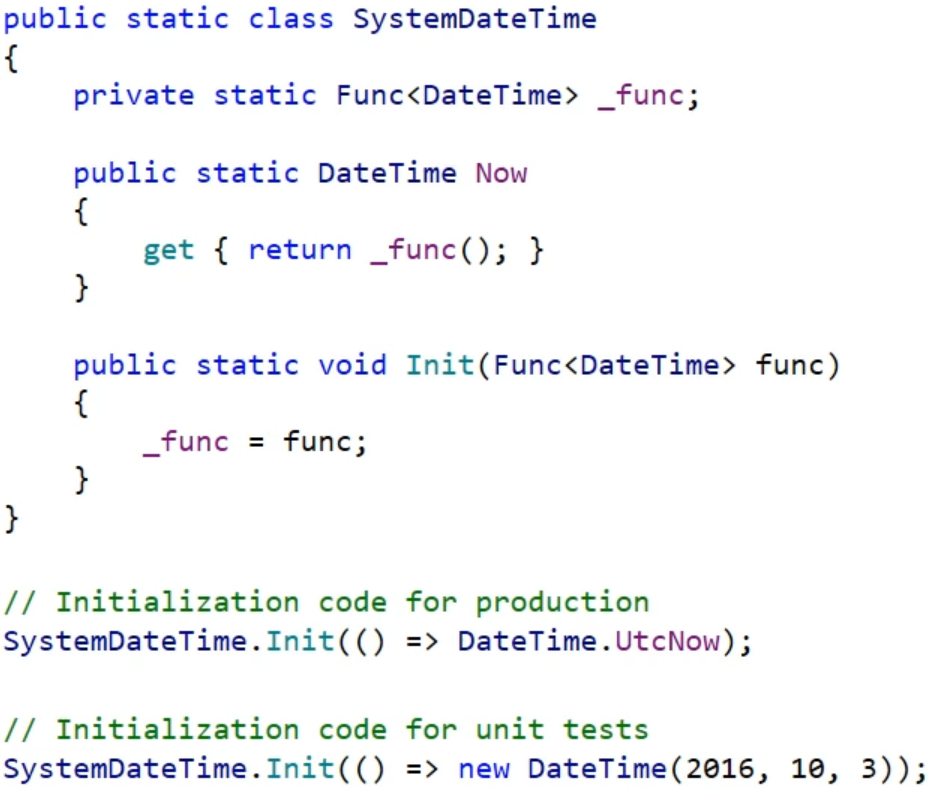
Use of Tread.Sleep() or Task.Delay() is almost a bad idea because we can not know for sure when the job will complete. And it takes excessive amount of time.

How to deal with asynchrony in test?

* Avoid testing asynchronous code as much as possible
  + You need to separate codes that deal with asynchrony and code that does the actual work and then test only the letter piece (the idea behind the Humble object pattern) and it can be applied broadly in many similar situations

Work with time?

* Inject the current date and time into the SUT
  + Works in simple scenarios
  + Cumbersome when passing it through multiple layers
* Introduce a simple static DateTime class and agree upon using this class in place of standard System.DateTime.Now of .Net
  + It allows us to initial it with a method that returns the current date and time



[Unit testing patterns for concurrent code](https://vimeo.com/171317257)