TDD & Clean Architecture Driven by Behaviour

Valentina Cupać - Optivem

About the speaker

Valentina Cupać coaches development teams in TDD & Clean Architecture to increase quality, accelerate delivery and scale teams.

Previously, she worked as a Senior Developer, Technical Lead & Solutions Architect.

Graduated from University of Sydney - Computer Science, Maths and Finance.

I write regular posts on LinkedIn about TDD & Clean Architecture.

Connect with me or **follow** me on LinkedIn:

https://www.linkedin.com/in/valentinacupac/

Agenda

1. Why are we here - TDD is painful, but is there another way?
2. The Deeper Why - Don't ship code, solve business needs
3. Executable Specifications - Do tests codify requirement specs or impl. specs?
4. What's a Unit Test? - Are we testing module behaviour or class structure?
5. Testing Behaviour - Tests should be coupled to behaviour, not to structure
6. TDD vs TLD - How do we drive development through executable requirements?
7. TDD & Clean Architecture - Driving architecture through system behaviour

Why are we here

TDD is painful, but is there another way?

Misconception #1 - The class is the unit of isolation

Write a **test class** for each **production class**.

Write **test method(s)** for each **production method**.

Isolate the class under test by **mocking** out all its collaborators.

Wikipedia says that unit testing means testing "**individual units of source code**", and in the case of OOP that we're testing a "**class**, or an individual **method**". We trust Wikipedia... right?
https://en.wikipedia.org/wiki/Unit_testing

Misconception #2 - Unit Tests must be expensive

It’s normal for **test code** to be **2-4X larger** than production code.

It’s normal that writing unit tests **takes up so much time**.

It’s normal that **unit tests break** when we **refactor** class design.

Anything that’s worthwhile must be painful. No pain, no gain, right?

Misconception #3: BDD is about behaviour, TDD isn't

ATDD and BDD are about behaviours. They are about testing our system from the user's perspective.

TDD is not about system behaviour, it's about testing classes and their interactions with other classes.

When we're under pressure and when the budget is tight, let's just keep ATDD/BDD. It actually tells us if we satisfied user requirements.

But what if we could solve the pains of TDD?

Imagine if TDD could really **speed up development**?

Imagine if TDD could be done with **significantly less test code**?

Imagine **if tests wouldn't break all the time** whilst you refactor your class designs?

Imagine if you could **test requirements at the unit level** and get really fast feedback?

Imagine if anyone - and **not just companies with huge budgets** - could get the benefits of TDD?

The Deeper Why

Our job is not to ship code, our job is to solve business needs

The Why

Why are we building houses? **To** have a place to live.

Why are we building cars? **To** be able to travel.

Why are we building software? **To** satisfy user needs.

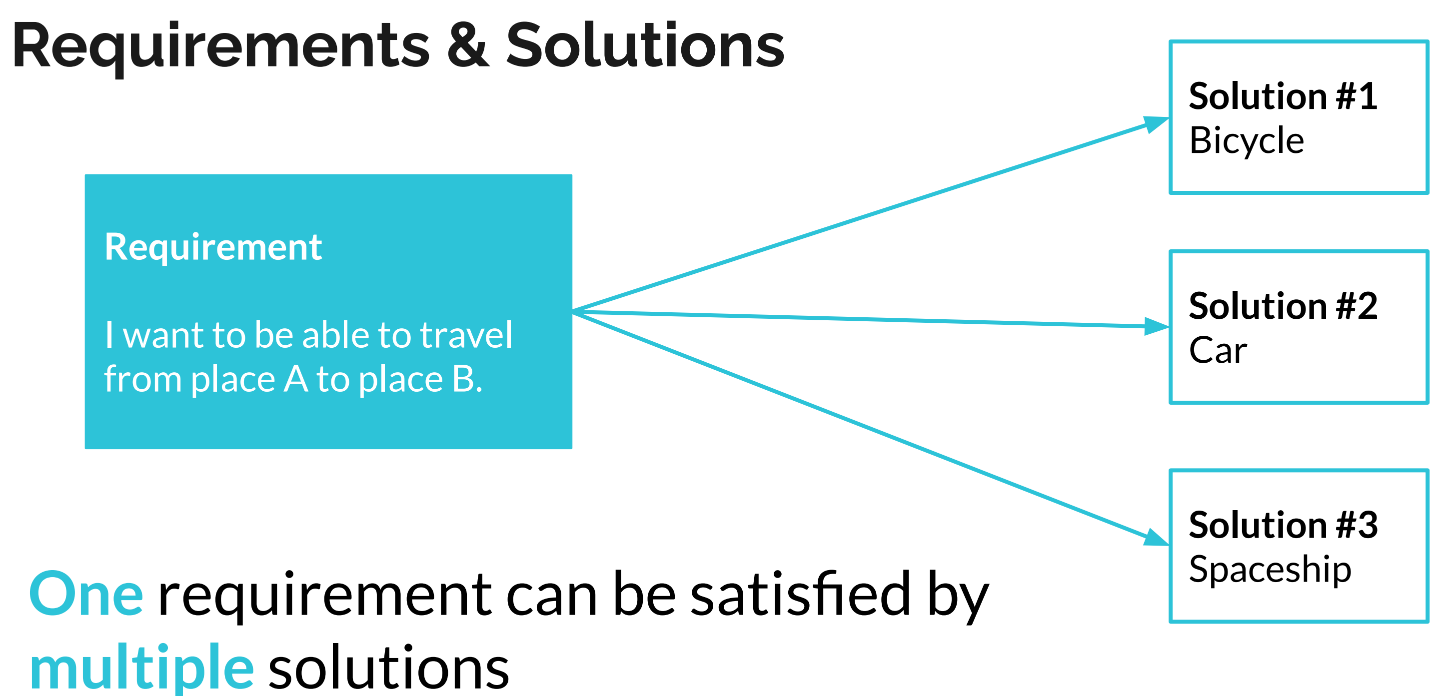
The Why

We don't get paid to "write code".

We get paid to solve business **needs**.

How? By converting **requirements** into software **solutions** to solve the business needs.

Requirements & Solutions



One requirement can be satisfied by multiple solutions

Tests as Executable Specifications

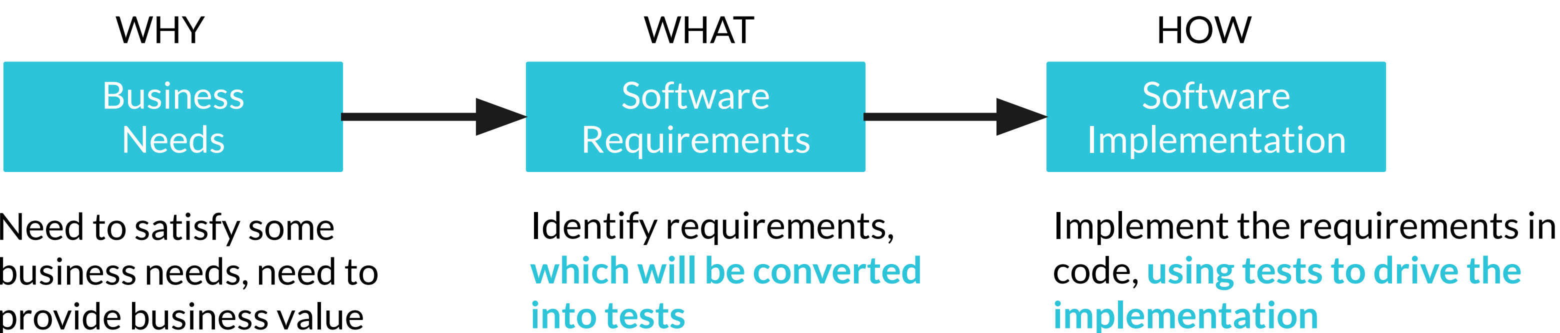
Do tests codify requirement specs or implementation specs?

Audience Poll

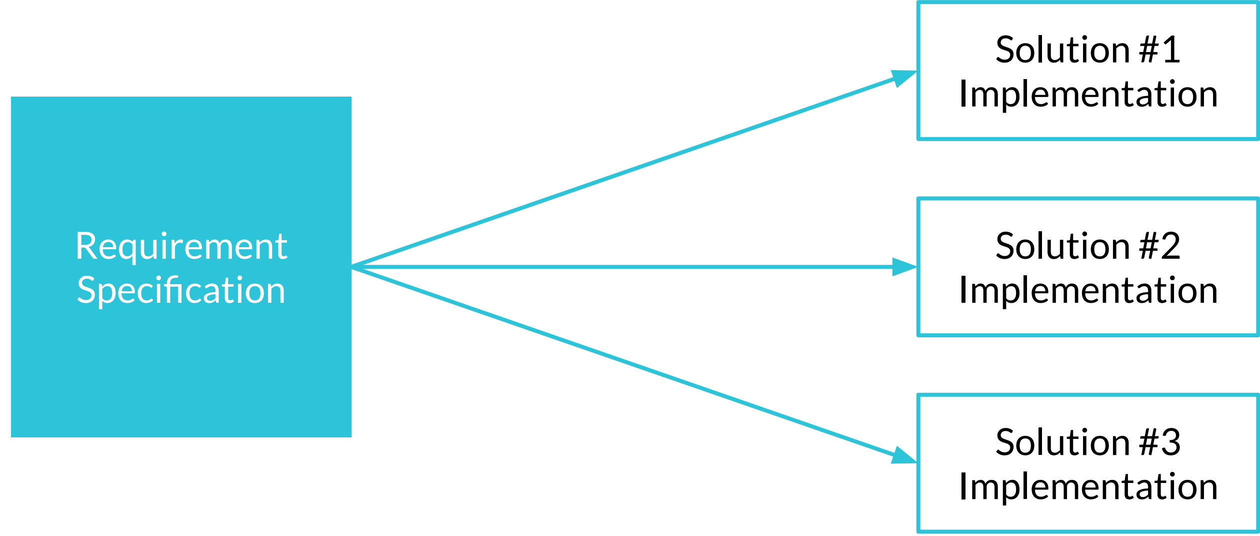
Are you familiar with the term “executable specifications”?

1. Yes
2. No
3. Sort of

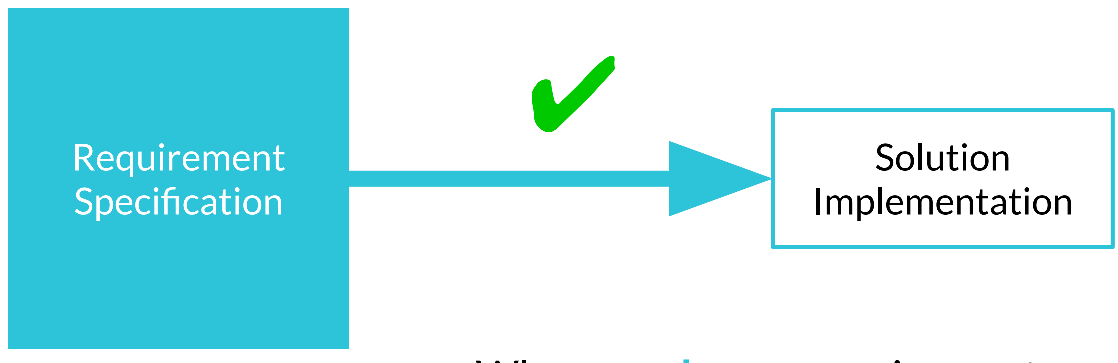
Requirements drive implementation



Requirements & Implementation

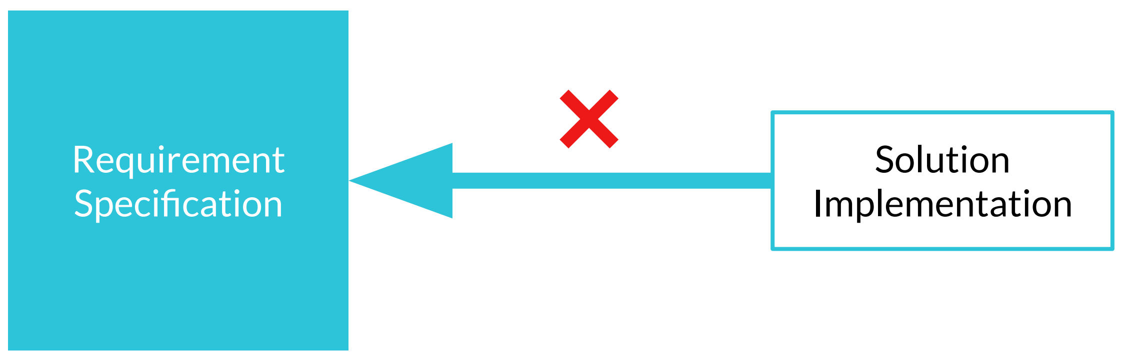


Requirements naturally affect implementation



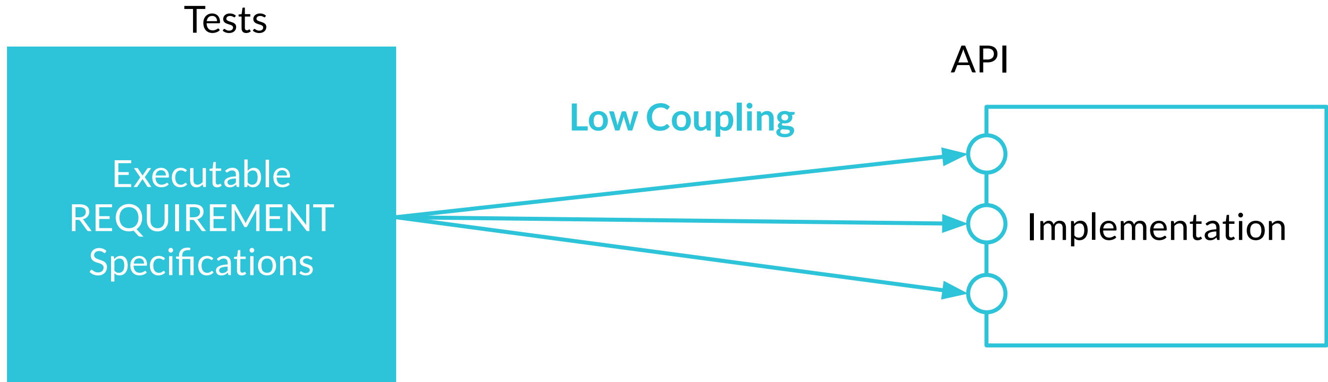
When we change requirements specifications, naturally we have to change the solution implementation too

Implementation should not affect requirements



When we **refactor or redesign** the solution implementation, it should **not change** the requirement specification

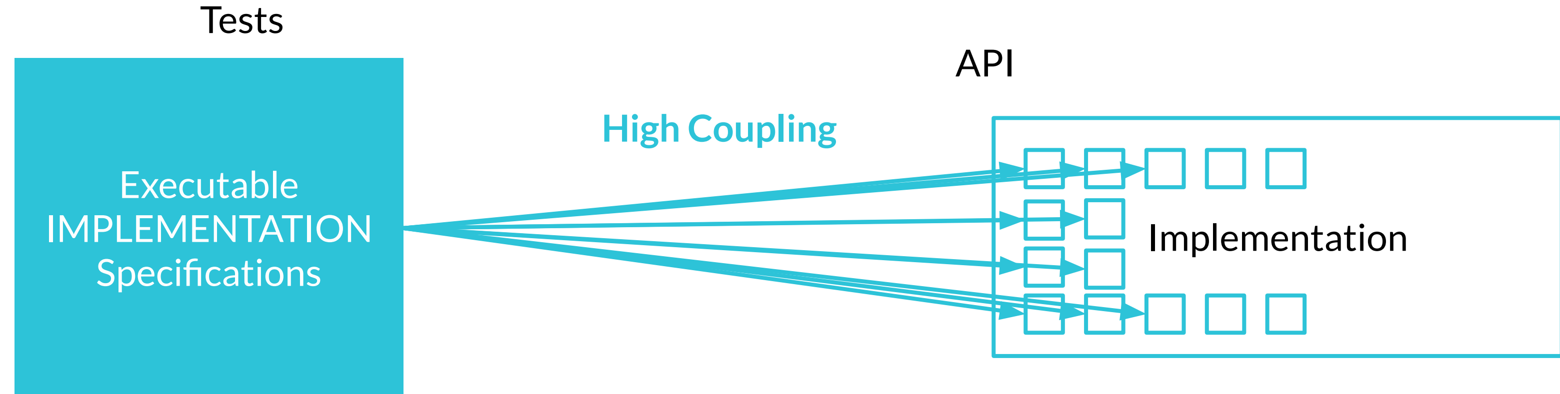
Tests as Requirement Specifications



Tests are coupled to the **API**, the **external behaviour**.

Robust tests - safely change the internal implementation without changing tests. Tests are changed only when the requirements change.

Tests as Implementation Specifications



Tests are coupled to the **implementation**, the **internal structure**.

Fragile tests - changing implementation breaks existing tests, causing tests to change even though requirements were not changed!

Summary - Testing requirements or design?

	Test = Requirement Spec	**Test = Implementation Spec**
Test coupling	Coupling to API	Coupling to Implementation
Test robustness	Robust tests	Fragile tests
Refactoring safety	Tests are stable	Tests break
Refactoring cost	No changes to tests	Tests have to be changed
ROI	High	Low

What's a Unit Test?

Are we testing module behaviour or class structure?

Audience Poll

What’s your familiarity with social vs solitary unit tests?

1. Didn’t hear about it
2. Heard about it, but not clear
3. Fully familiar with it

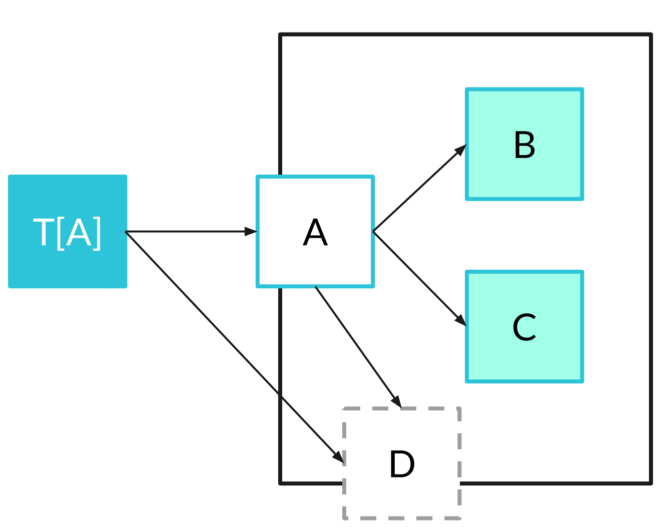
What's a Unit Test?

- Verifies a **unit**
- Verifies it in **isolation**
- Verifies it **quickly**

	Sociable Unit Tests (Classical TDD)	**Solitary Unit Tests (Mockist TDD)**
Unit	One module *(one or more classes) (coarse-grained)*	One class *(fine-grained)*
Isolation	Isolate module ONLY from shared dependencies (DB, Files, etc.)	Isolate class from ALL its collaborators

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

Sociable Unit Tests - Testing Module API



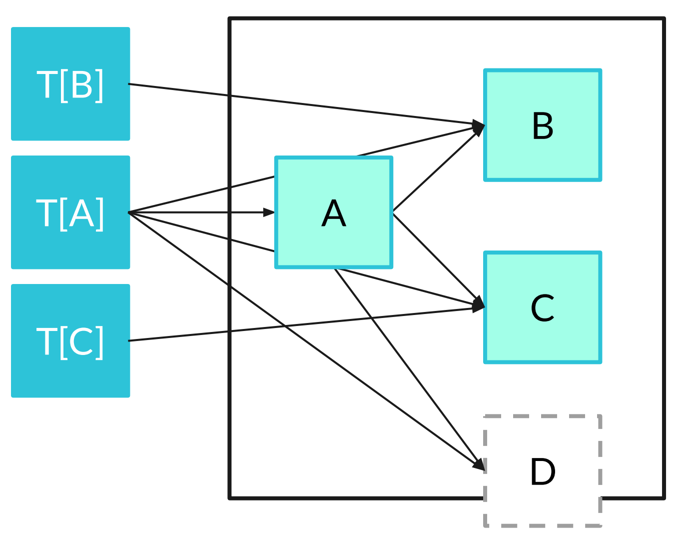
Sociable unit tests access the module API.

They don't know about the module's implementation details.

We use test doubles only for shared dependencies (DB, Files, etc.)

→ Refactoring module's implementation has no impact on tests. ✔

Solitary Unit Tests - Testing Module Implementation



Solitary unit tests access the module implementation.

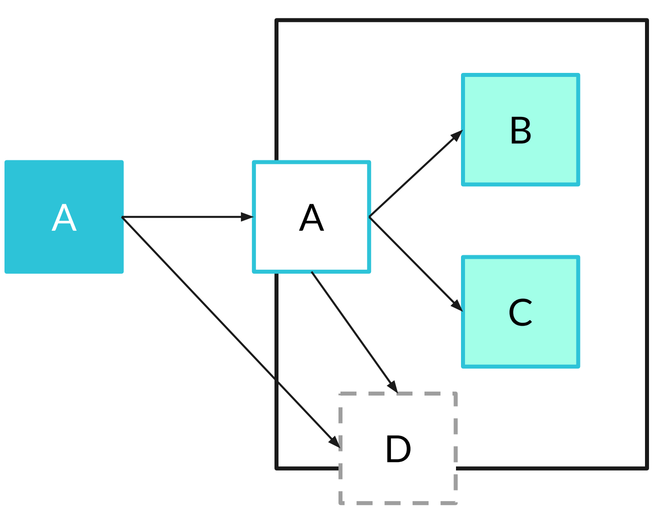
They know about module's internal classes and their collaborators.

We mock all the collaborators.

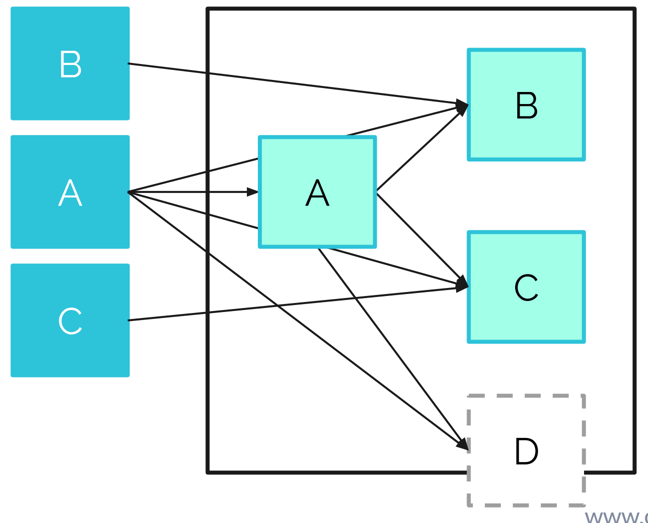
→ Refactoring the module's implementation breaks existing tests.

Micro Comparison

Sociable Unit Tests



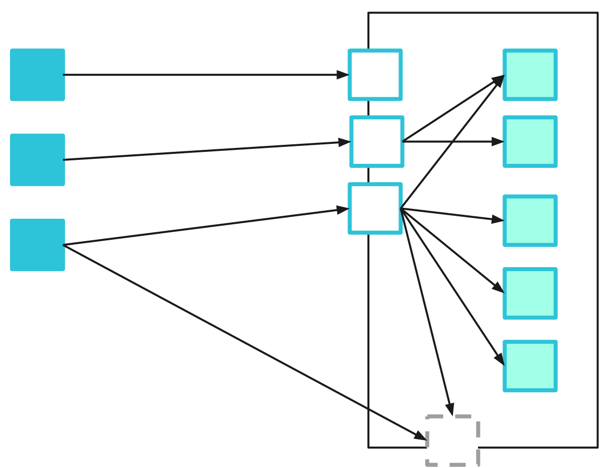
Solitary Unit Tests

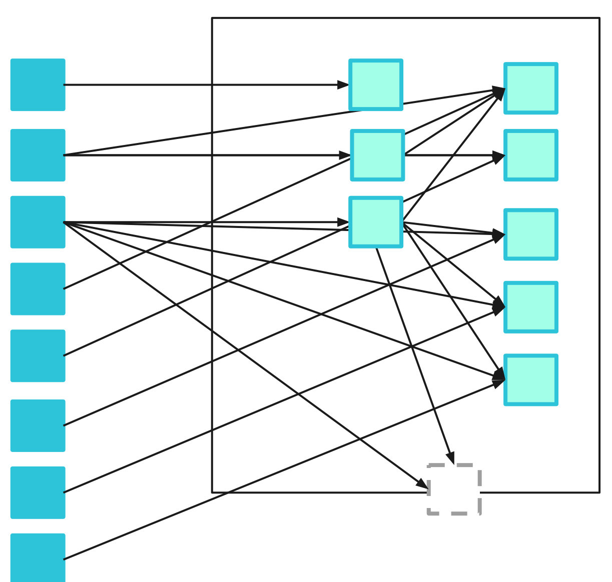


Macro Comparison

Solitary Unit Tests

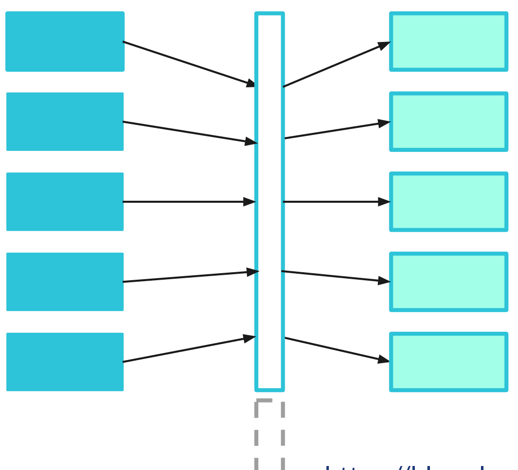
Sociable Unit Tests



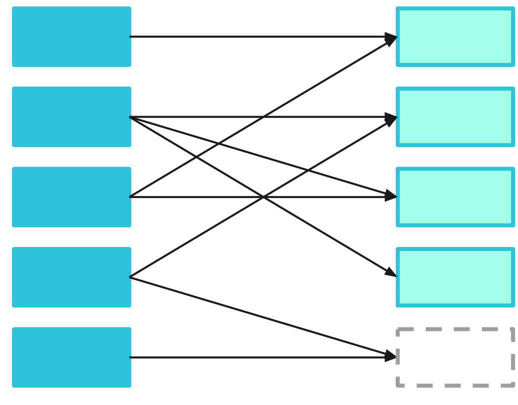


Macro Comparison II

Social Unit Tests
Tests coupled to API



Solitary Unit Tests
Tests coupled to Implementation



https://blog.cleancoder.com/uncle-bob/2017/03/03/TDD-Harms-Architecture.html

Unit Test Comparison

Sociable Unit Tests (coarse-grained)	Solitary Unit Tests (fine-grained)
Tests are coupled to module API (module behaviour)	Tests are coupled to module implementation (module structure)
Robust tests → Refactoring module implementation does not impact tests	Fragile tests → Refactoring module implementation causes tests to break
Lower cost → Less test code, higher test stability, lower maintenance cost	Higher cost → More test code, lower test stability, higher maintenance cost

Testing Behaviour

Tests should be coupled to behaviour, not to structure

Audience Poll

What are the origins of TDD and BDD?

1. TDD was originally about tests, and BDD was originally about behaviour
2. Both TDD and BDD were originally about behaviour
3. Not really sure

Kent Beck - Tests should be coupled to behaviour

Programmer tests should be **sensitive to behavior changes** and **insensitive to structure changes**. - Kent Beck

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

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

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

Tests should be **coupled to the behavior** of code and **decoupled from the structure** of code. - Kent Beck

https://twitter.com/kentbeck/status/1182714083230904320?lang=en

Dan North - Behaviour Driven Development (BDD)

"**Behaviour**" is a more useful word than "**test**" - Dan North

Requirements are **behaviour** - Dan North

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

Dan North attempted to "fix" the naming confusing by replacing the word "test" by "behaviour". Even though many people associate BDD with ATDD/Gherkin/Cucumber, the origins of BDD were actually an attempt to showcase the behavioural intention of TDD.

Martin Fowler - Refactoring

Refactoring is a disciplined technique for **restructuring** an existing body of code, altering its internal structure **without changing its external behavior** - Martin Fowler

https://martinfowler.com/tags/refactoring.html

When we refactor, we change structure but not behaviour!

Testing at Google - “Striving for Unchanging Tests”

*“... the ideal test is **unchanging**...”*

“When an engineer **refactors the internals** of a system without modifying its interface… the system’s **tests shouldn’t need to change**. The role of tests in this case is to ensure that the refactoring didn’t change the system’s behavior.”

“**Changing a system’s existing behavior** is the one case when we expect to have to make **updates to the system’s existing tests**.”

https://www.amazon.com/Software-Engineering-Google-Lessons-Programming-ebook-dp-B0859PF5HB/dp/B0859PF5HB

Testing at Google - “Test via Public APIs”

“... let’s look at some practices for making sure that tests don’t need to change unless the requirements of the system being tested change.”

“By far the most important way to ensure this is to **write tests** that would **invoke the system** being tested **in the same way its users would**; that is, making calls against its **public API** rather than implementation details.”

“If **tests work the same way as the system’s users**, by definition, change that breaks a test might also break a user.”

https://www.amazon.com/Software-Engineering-Google-Lessons-Programming-ebook-dp-B0859PF5HB/dp/B0859PF5HB

Testing at Google - “Test Behaviors, Not Methods”

“The **first instinct** of many engineers is to try to **match the structure** of their tests to the structure of their code such that every production method has a corresponding test method.”

“This pattern can be convenient at first, but **over time it leads to problems**.”

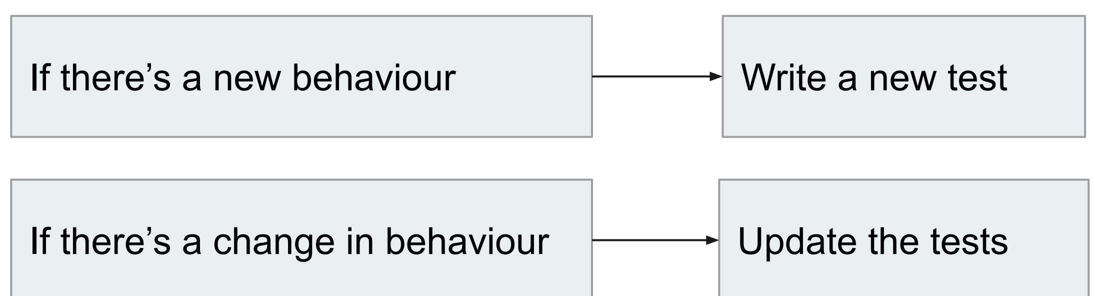
“There’s a better way: rather than writing a test for each method, **write a test for each behavior**.”

https://www.amazon.com/Software-Engineering-Google-Lessons-Programming-ebook-dp-B0859PF5HB/dp/B0859PF5HB

When to write new tests or change tests?

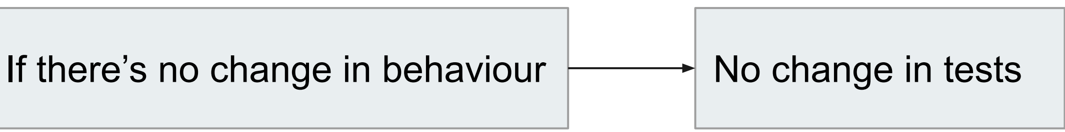
BEHAVIOURAL CHANGES

New or changed
business requirements



STRUCTURAL CHANGES

Refactoring or redesign



TDD vs TLD

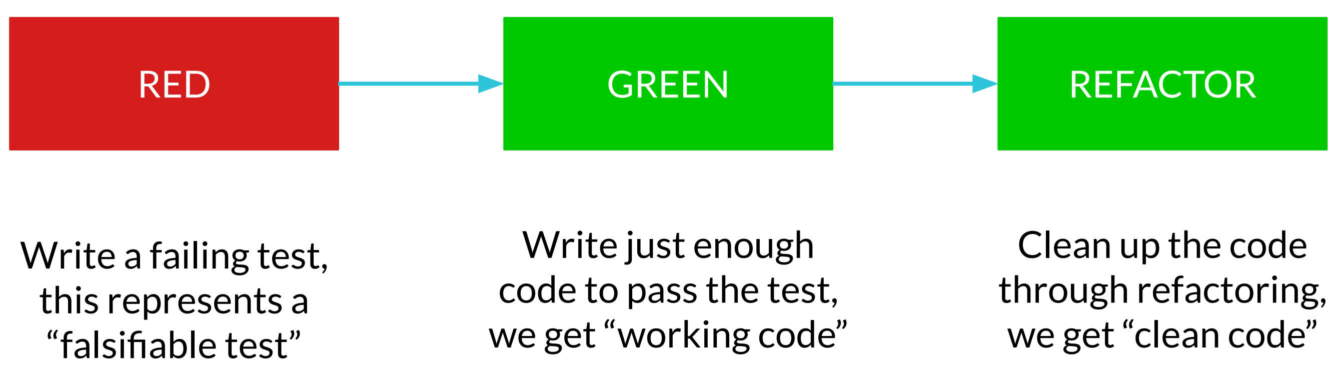
How do we drive development through executable requirements?

Audience Poll

Did you try TDD? What was your experience with TDD?

1. Never tried TDD
2. Tried TDD, but not convinced
3. Tried and partially picked up TDD
4. Tried and fully adopted TDD

TDD Red-Green-Refactor



TDD Feedback Loops

1. **REQUIREMENT TESTABILITY:** Can we write a test for the requirement?
2. **TEST FALSIFIABILITY:** Do we see the test fail? The RED step.
3. **INTERFACE DESIGN:** Is the interface user-friendly? The test is the first consumer.
4. **IMPLEMENTATION CORRECTNESS:** Does the code work? The GREEN step.
5. **IMPLEMENTATION QUALITY:** Is the implementation clean? The REFACTOR step.

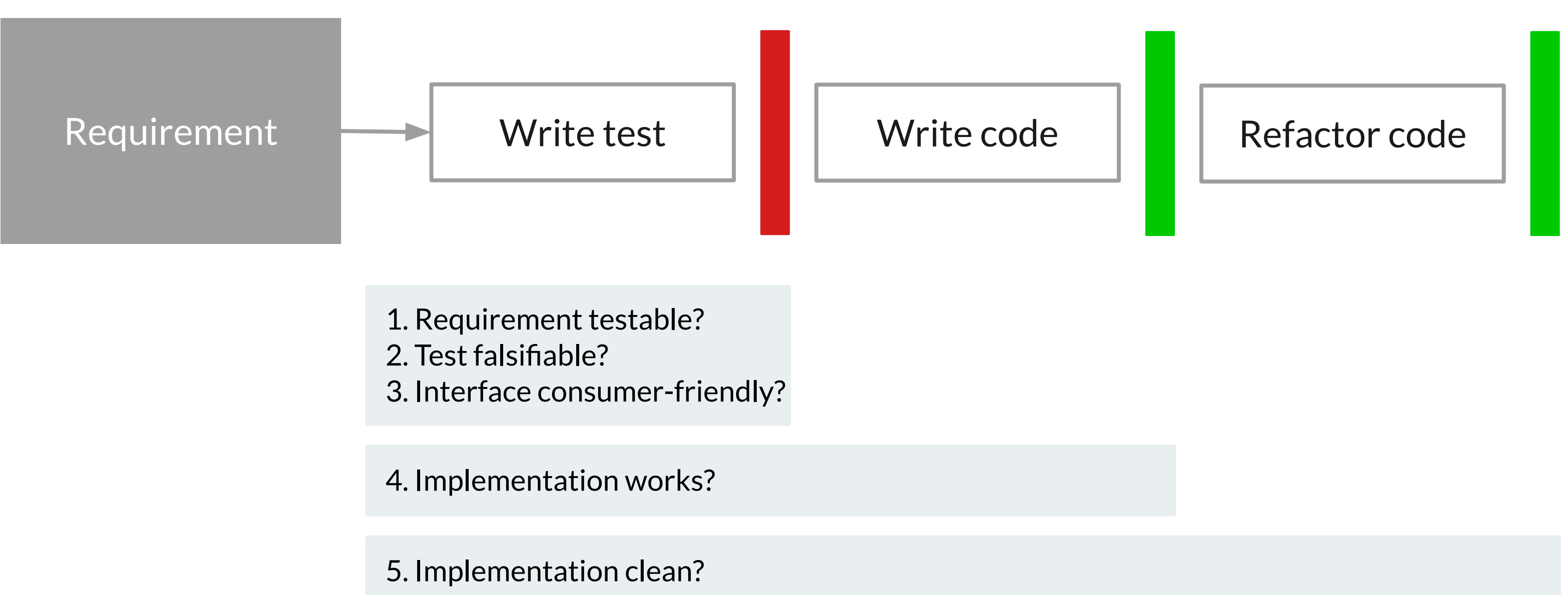
Audience Poll

When does your team write unit tests?

1. We don’t write unit tests at all because my team doesn’t want to
2. We don’t write unit tests because we don’t have the budget/time for it
3. We firstly write code, then write the unit tests afterwards
4. We always write the unit test first, then write code after the test

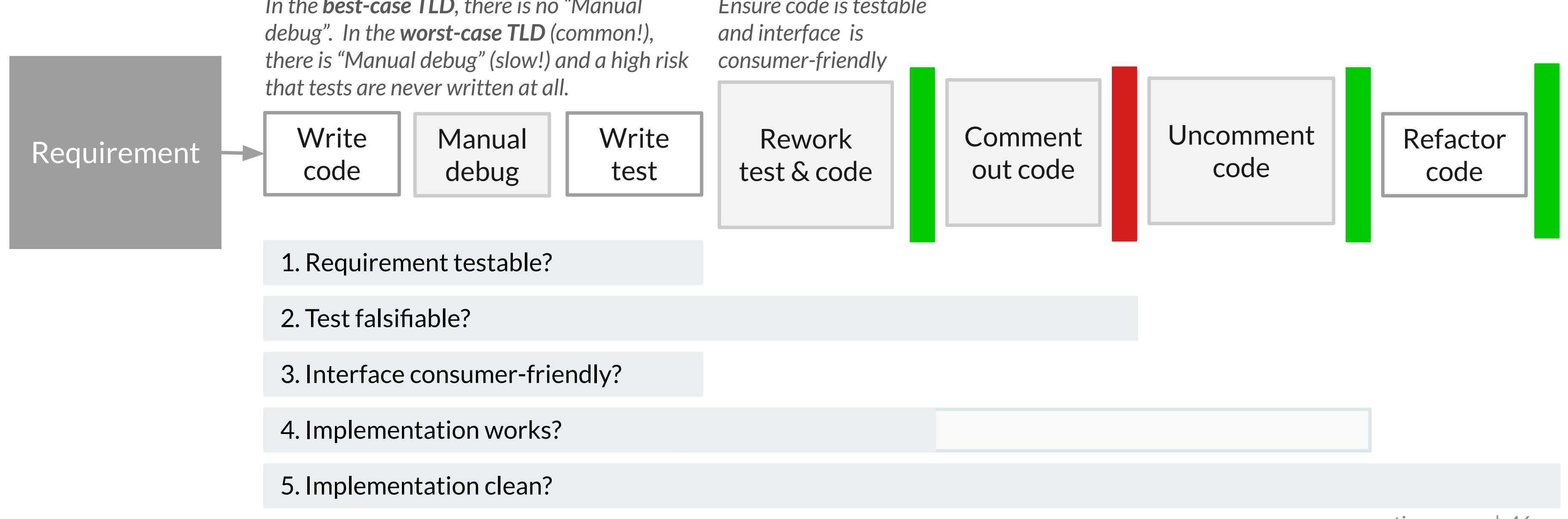
Test Driven Development

TDD results in faster development due to shorter feedback loop



Test Last Development

TLD results in slower development due to longer feedback loop



TDD vs TLD - Summary

TDD results in faster development due to shorter feedback loop

TDD guarantees that code is covered by tests (because we never write code without tests first)

TLD results in slower development due to longer feedback loop

TLD does not guarantee that code will be covered by tests (in the worst case, tests may never be written)

TDD & Clean Architecture

Driving application architecture through system behaviour

Audience Poll

Does your team use any of these architectures?

1. CRUD - Controllers, Services, Entities (ORM), Repositories (ORM)
2. Hexagonal Architecture
3. Onion Architecture
4. Clean Architecture
5. Something else

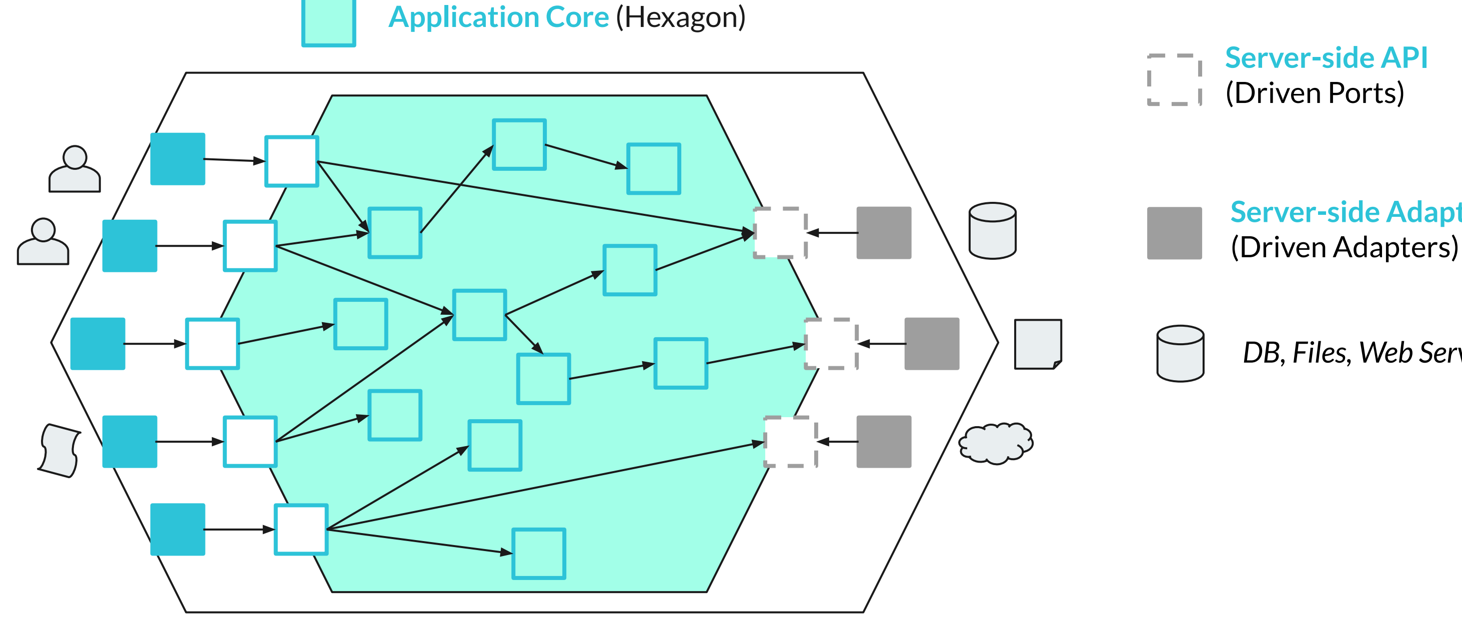
https://alistair.cockburn.us/hexagonal-architecture/

Hexagonal Architecture

Users, Programs, Scripts

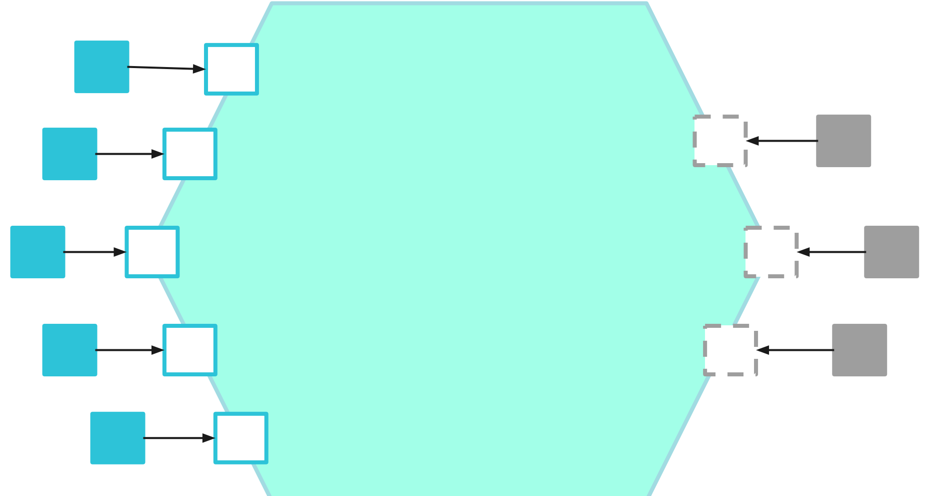
User-side API (Driver Ports)

User-side Adapters (Driver Adapters) *GUI, Console, REST API, Tests*



Hexagonal Architecture - Unit Testing

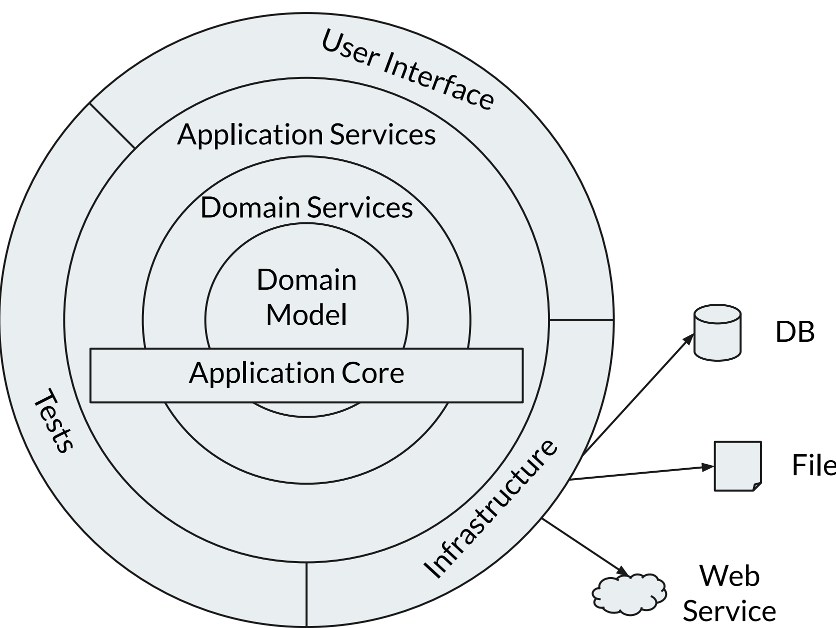
Unit Tests can execute system use cases through the **user-side API**



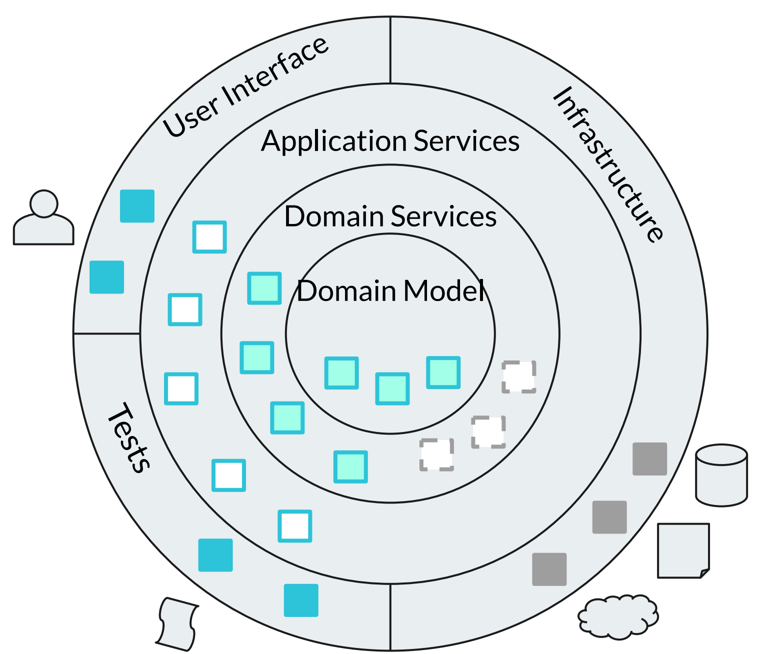
Test Doubles serve as in-memory adapters for the **server-side API**

Onion Architecture

https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/



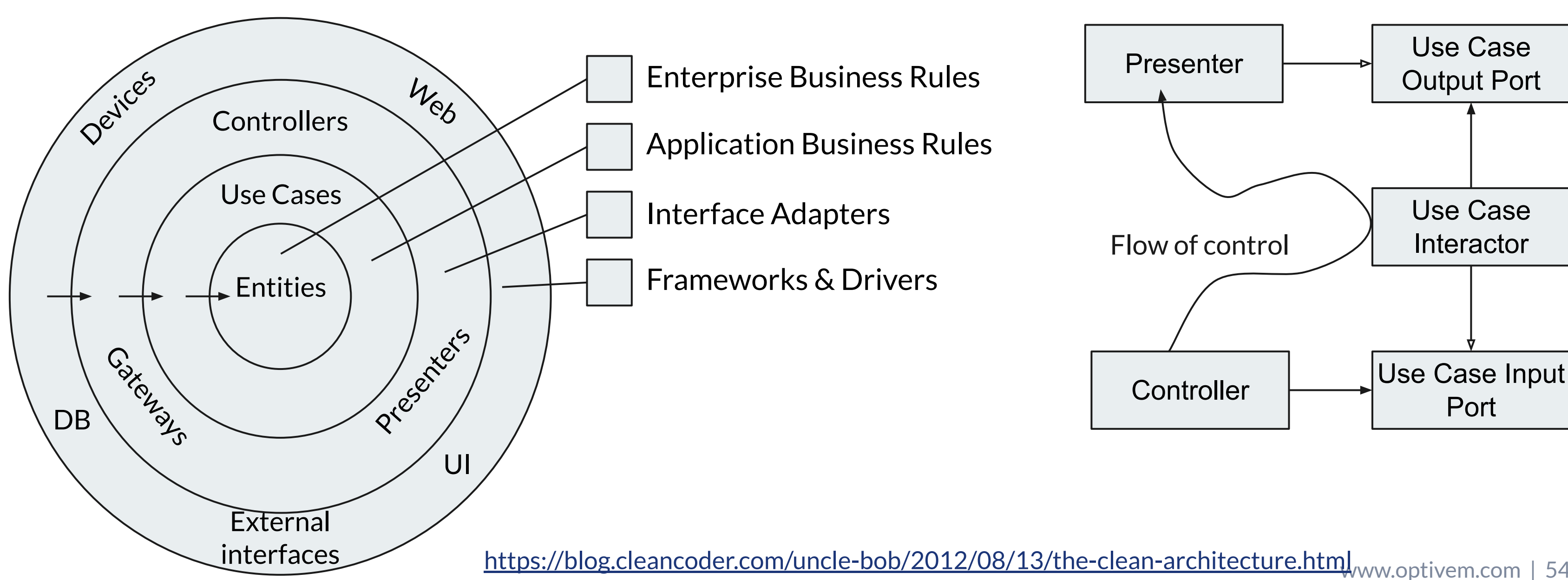
Onion Architecture & Hexagonal Architecture



Hexagonal Architecture

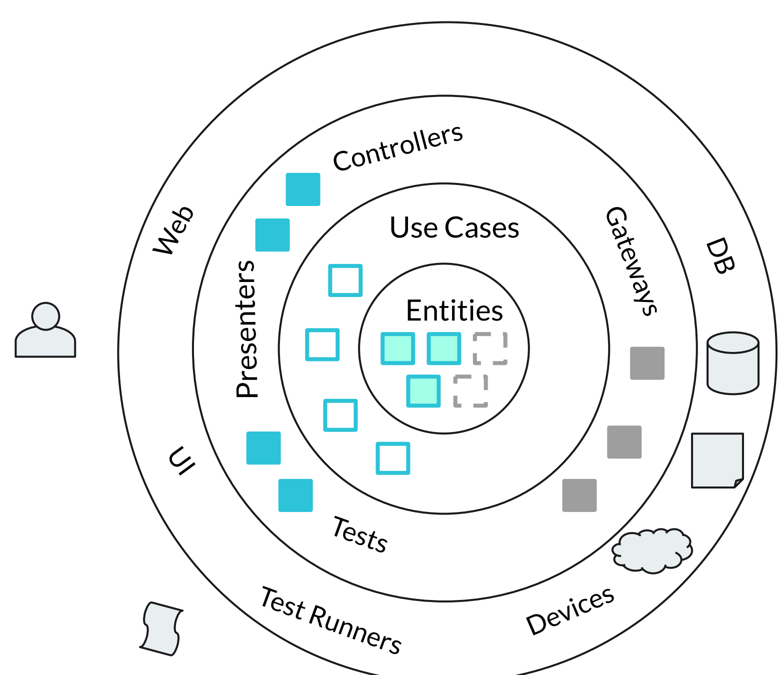
- Users
- User-side Adapters
- User-side Ports
- Application Core
- Server-side Ports
- Server-side Adapters
- Technologies

Clean Architecture



https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Clean Architecture & Hexagonal Architecture



Hexagonal Architecture

- Users
- User-side Adapters
- User-side Ports
- Application Core
- Server-side Ports
- Server-side Adapters
- Technologies

Architectural Equivalence

System under Test (SUT)	Hexagonal Architecture	Onion Architecture	Clean Architecture
Users	Users	Users	Users
SUT Tests	User-side Adapters	UI, Tests	Presenters, Tests
SUT API	User-side Ports	Application Services	Use Cases
SUT Implementation	Application Core	Domain Model	Entities
Shared Dependency Interfaces	Server-side Ports	Domain Services	Gateway Interfaces
Shared Dependency Implementations	Server-side Adapters	Infrastructure	Gateways
Technologies	Technologies	Technologies	Technologies

Acceptance Testing - Tests acting as the Users

Acceptance Testing - Unit Level

Unit Tests execute use cases

Shared dependencies are substituted with test doubles

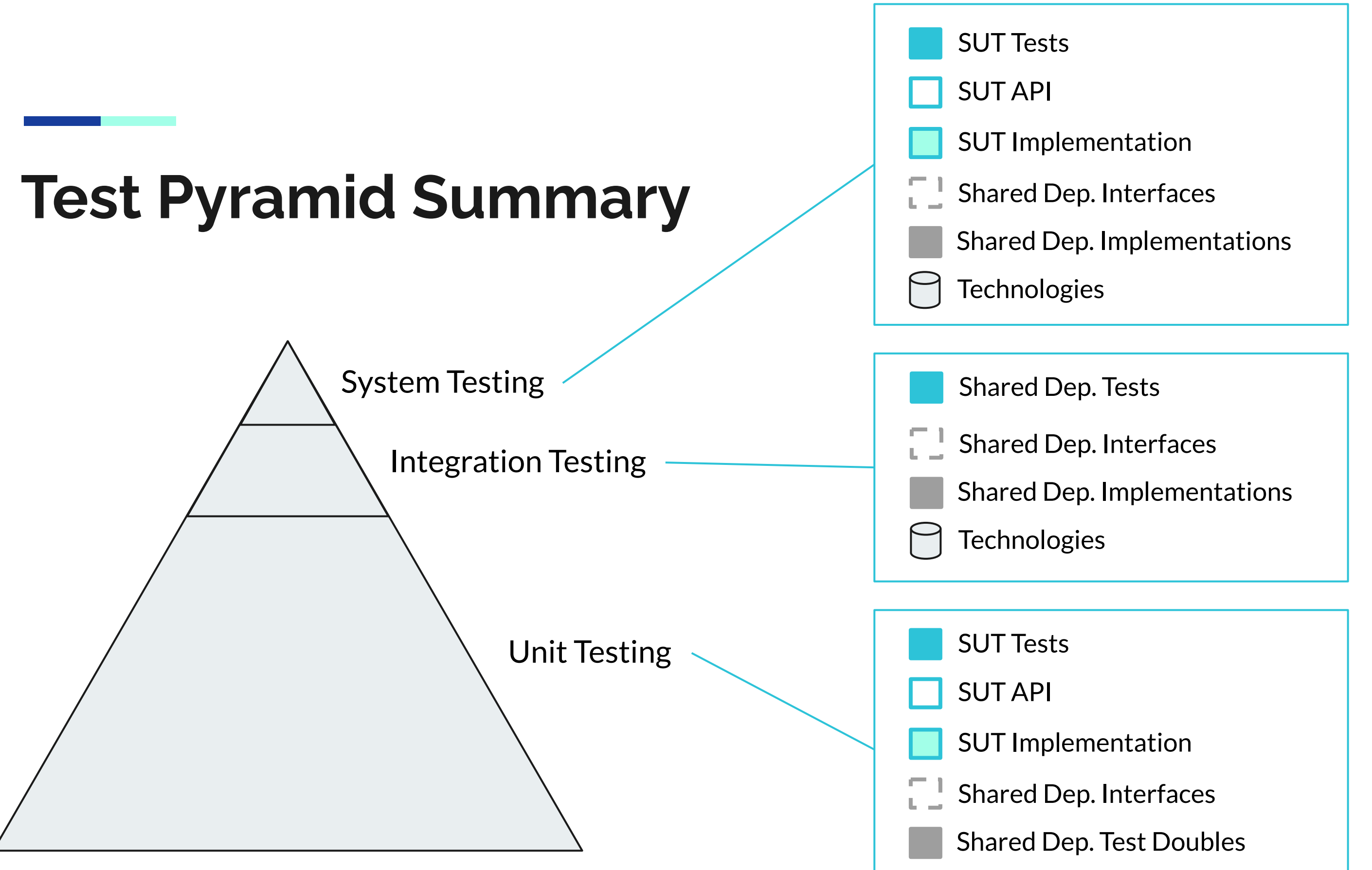
Acceptance Testing - E2E Level

UI Automation runners execute use cases

Shared dependencies are substituted with real implementations

Benefit: we can run acceptance tests at the unit level through the use case ports, like the user!
Much faster feedback & scenario coverage at the unit level

Test Pyramid Summary



Conclusion

Tests should be executable requirement specs… not implementation specs

Tests should be coupled to the API… not the implementation

Tests should be coupled to behaviour… not to structure

Clean Architecture exposes use cases, we can test the application behaviour

Refactoring does not change behaviour, does not affect behavioural tests

Behavioural tests are more robust and have lower test maintenance cost ✔

Thank You

Valentina Cupać @ Optivem

Connect or **follow** me on LinkedIn to learn more about TDD and Clean Architecture

https://www.linkedin.com/in/valentinacupac/

E valentina.cupac@optivem.com

W www.optivem.com