Photo by NASA on Unsplash

TDD and Clean Architecture
Use Case Driven Development

Valentina Cupać
Founder & Technical Coach @ 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 articles about TDD & Clean Architecture.

Connect with me or follow me:

Twitter: @ValentinaCupac
GitHub: https://github.com/valentinacupac
LinkedIn: https://www.linkedin.com/in/valentinacupac/

Agenda

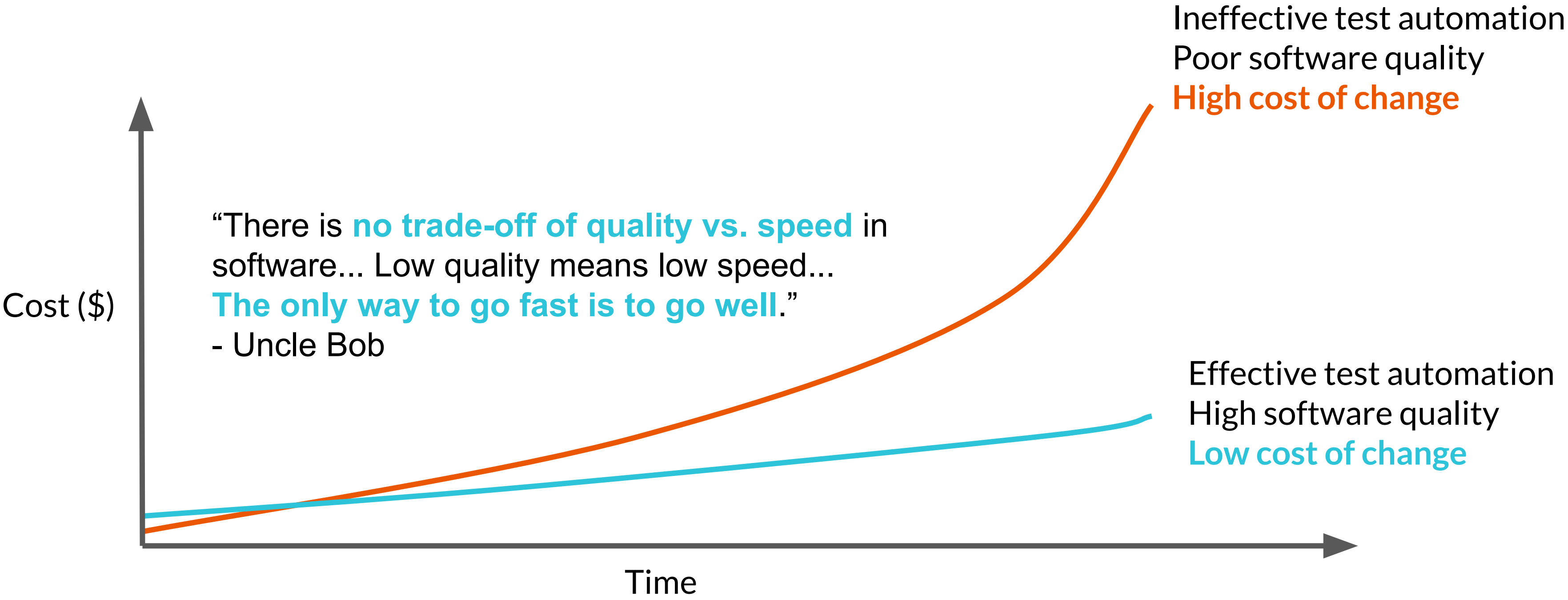
1. Economics of Quality - Effective unit tests reduce software maintenance costs
2. Modular Architecture - Designing systems with modular components and DI
3. Unit Testing - Coupling to the module interface, not the implementation
4. Clean Architecture - Hexagonal Architecture & Use Case Driven Development (UCDD)
5. Architecture Implementation - CRUD, Onion Architecture and Clean Architecture
6. Use Case Implementation - Behavioral decomposition of Use Cases and the Domain
7. Use Case Unit Testing - Approaches to Unit Testing Use Cases and the Domain
8. GitHub Code Demo - TDD & Clean Architecture with UCDD (Java & .NET)

Photo by Grant Ritchie on Unsplash

1. Economics of Quality

Effective unit tests reduce software maintenance costs

Economics of Software Quality



"There is **no trade-off of quality vs. speed** in software... Low quality means low speed... **The only way to go fast is to go well**."
- Uncle Bob

Effective Unit Testing

Effective Unit Tests

- Economic to write and maintain
- Robust during refactoring
- Verifying requirements

How? Coupling tests to system behavior, system API (use cases).

Ineffective Unit Tests

- Expensive to write and maintain
- Fragile during refactoring
- Failing to verify requirements

How? Coupling tests to system implementation (internal system classes).

Effective Tests - Kent Beck & Dan North

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

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

Requirements are **behaviour** - *Dan North*

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

Effective Tests - Software Engineering at Google

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. - *Software Engineering at Google*

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. - *Software Engineering at Google*

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

Effective Tests - Driven by Behavior

During a previous meetup, we reviewed the difference between behavioral and structural approaches to testing. Driving tests with behavior helps lead to more robust tests which have lower maintenance costs.

TDD and Clean Architecture - Driven by Behaviour (Valentina Cupać)

Hosted by: Java User Group Switzerland & Software Crafts Romandie Community

YouTube Recording: https://www.youtube.com/watch?v=3wxiQB2-m2k

Effective versus ineffective tests

When you open up the project source code, does the directory listing show behaviors of the system (use cases)? Or do you have to dig through code to see the use cases?

Are your tests coupled to behaviors of the system, the system API, the use cases? Or are the tests coupled to the system structure, the system implementation, its classes and methods?

Are your tests executable requirement specifications, the tests show the flows of use cases? Or are the tests executable implementation specifications, representing UML class diagrams?

Are your tests robust when you refactor system internals, are the tests cheap to maintain? Or are the tests fragile, breaking during refactoring, and expensive to maintain?

Why? Coupling tests to system behavior (use cases) instead of system structure (internal classes).

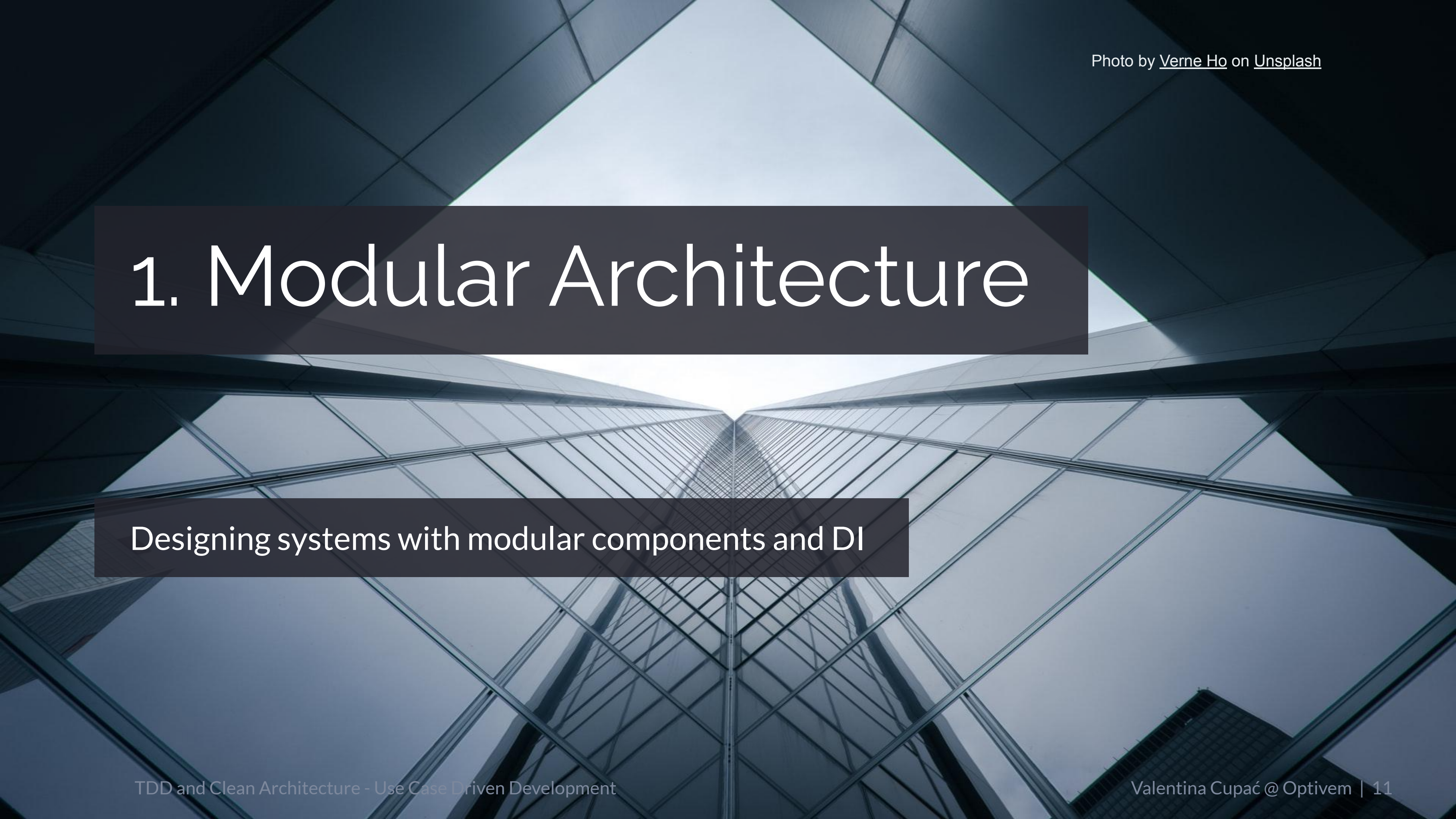


Photo by Verne Ho on Unsplash

1. Modular Architecture

Designing systems with modular components and DI

What's a Module? A Unit.

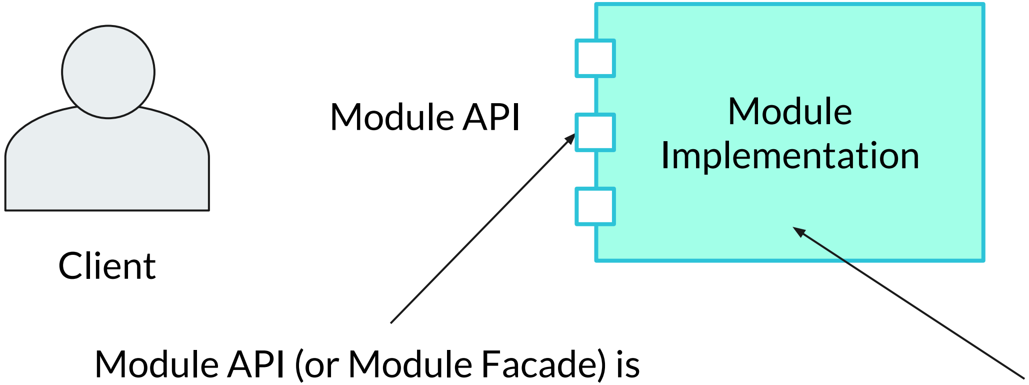
"A **module** is a separate **unit** of software or hardware."

https://www.techtarget.com/whatis/definition/module

Typical characteristics of **modular components** include **portability**, which allows them to be used in a variety of systems, and **interoperability**, which allows them to function with the components of other systems.

https://www.techtarget.com/whatis/definition/module

Module is the Unit



Module API (or Module Facade) is the external interface to the module, represents module behavior, visible to clients.

Module Implementation is the internal implementation of the module, represents module structure, hidden from clients.

Note: API refers to the public interface, does not refer to REST API

Modular Design

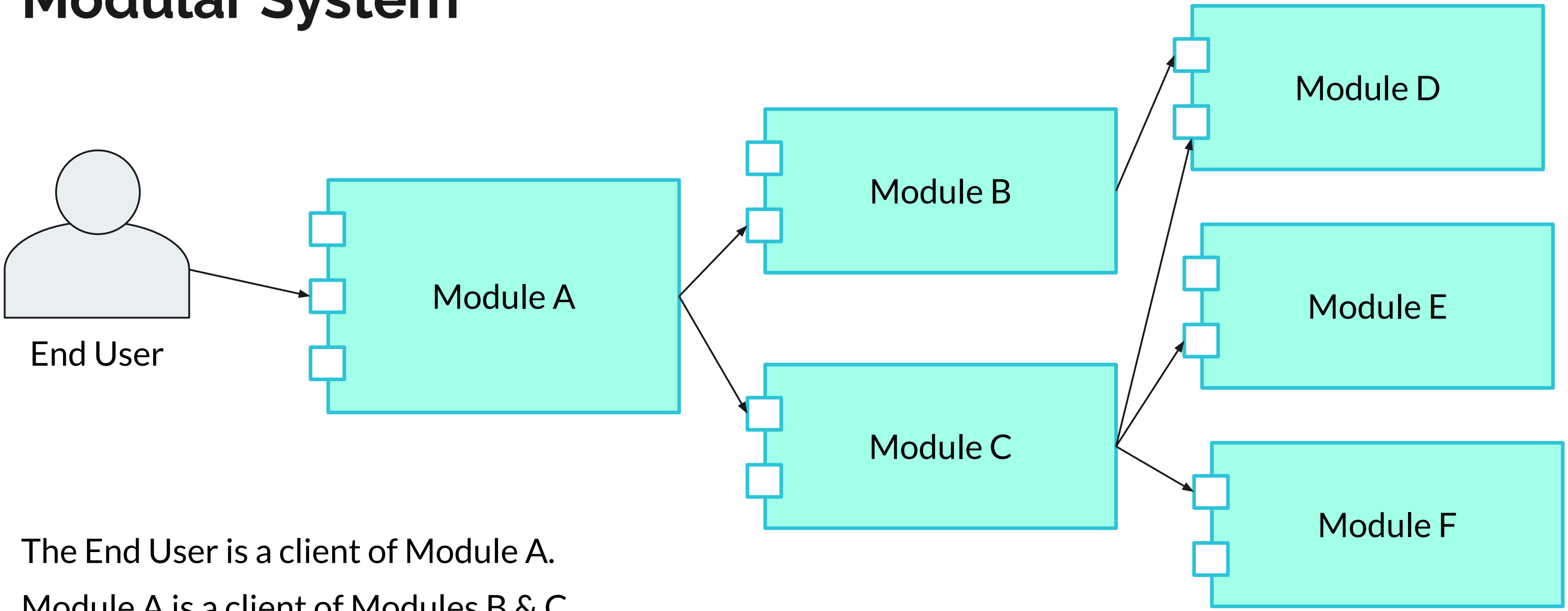
A module should have a **single purpose**, a single responsibility.

A module's **API** should be **consumer-friendly**, complete and minimal.

A module should **encapsulate its implementation details**, so that we can change the implementation details without affecting clients.

https://www.genui.com/resources/5-essential-elements-of-modular-software-design

Modular System



The End User is a client of Module A.

Module A is a client of Modules B & C.

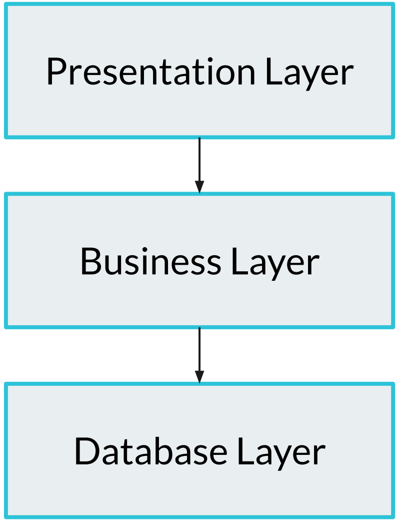
Dependency Inversion Principle

“A. High-level modules should not depend on low-level modules. Both should **depend on abstractions**.”

“B. Abstractions should not depend upon details. Details should **depend upon abstractions**.”

https://codecraft.medium.com/depend-on-abstractions-and-not-on-modules-with-the-dependency-inversion-principle-f31b9d10789b

Traditional Layered Architecture (without DI)



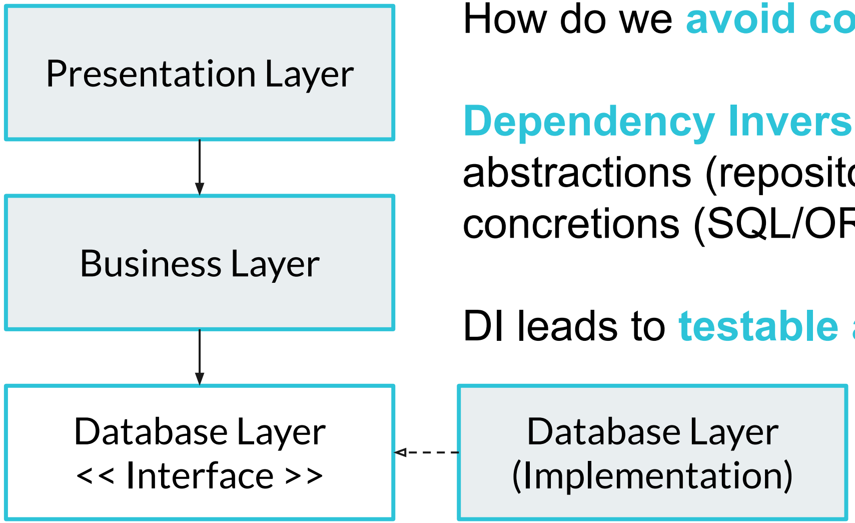
In the traditional layered architecture, each layer is dependent on the layer below it.

The Business Layer is **directly coupled** to the Database Layer. The Presentation Layer is **transitively coupled** to the Database Layer.

The **database** is at the **heart of the system**. Everything points towards the database.

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

Testable Layered Architecture (with DI)



How do we **avoid coupling** UI & BL to the DB?

Dependency Inversion (DI) Principle. Depend on abstractions (repository interfaces) instead of concretions (SQL/ORM implementation).

DI leads to **testable architecture**.

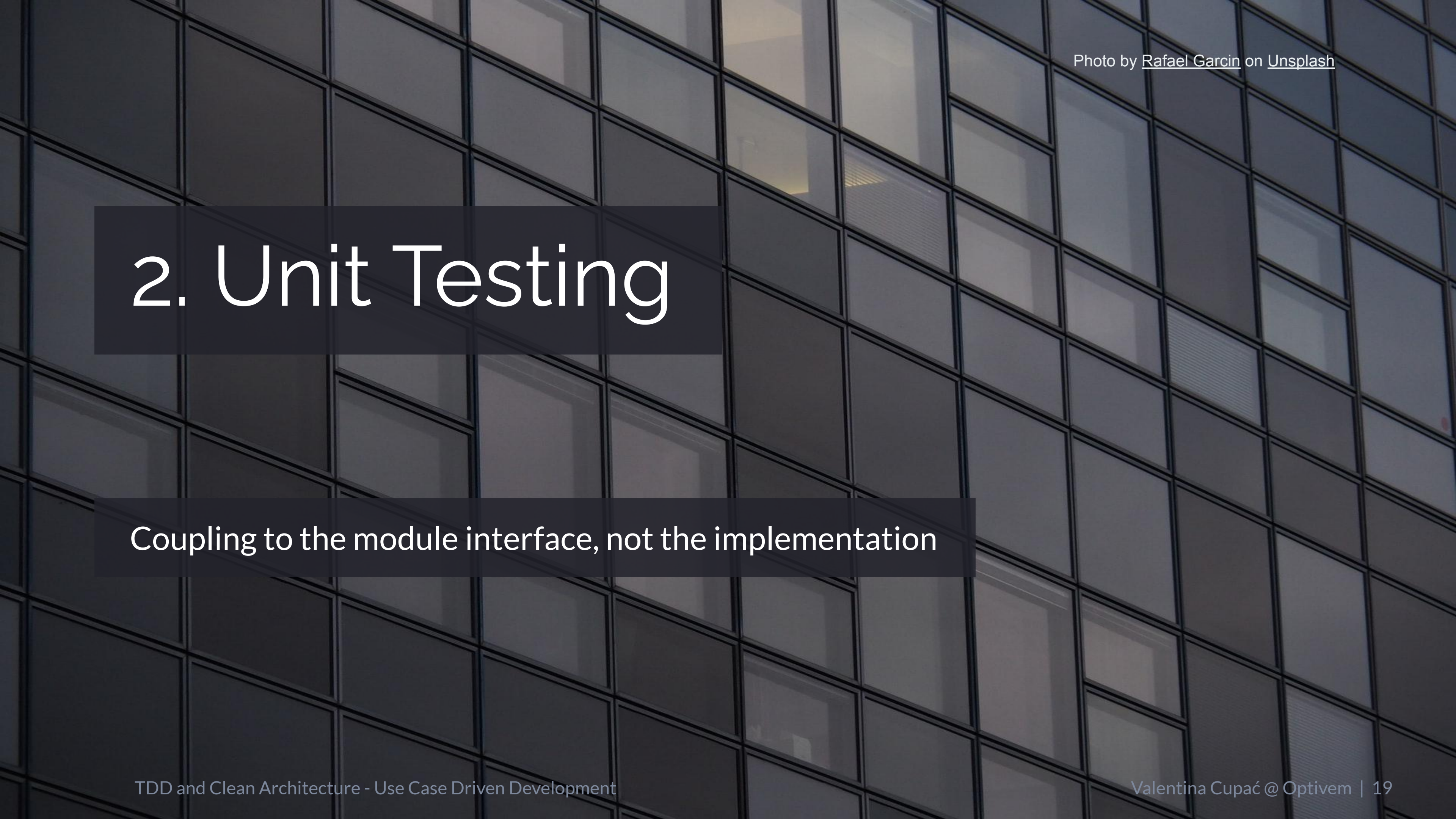
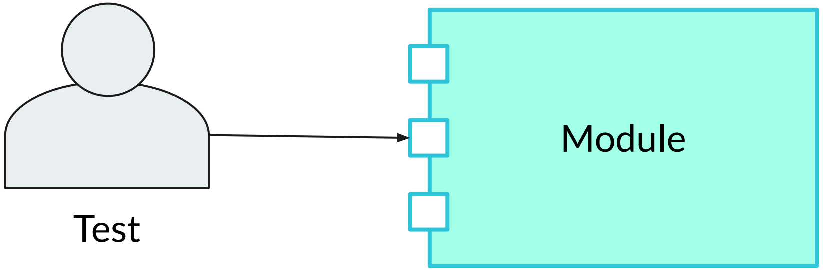


Photo by Rafael Garcin on Unsplash

2. Unit Testing

Coupling to the module interface, not the implementation

Module Testing = Unit Testing



The module is a unit of software.

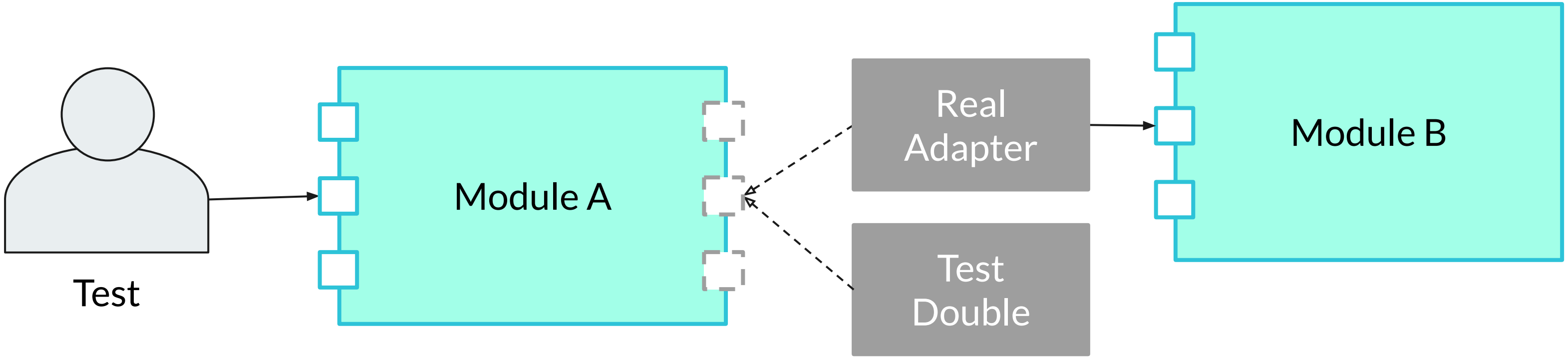
A module test is a unit test.

The test acts as a client of the module. It verifies the client requirement specifications.

The test is coupled to the module API, not the module implementation.

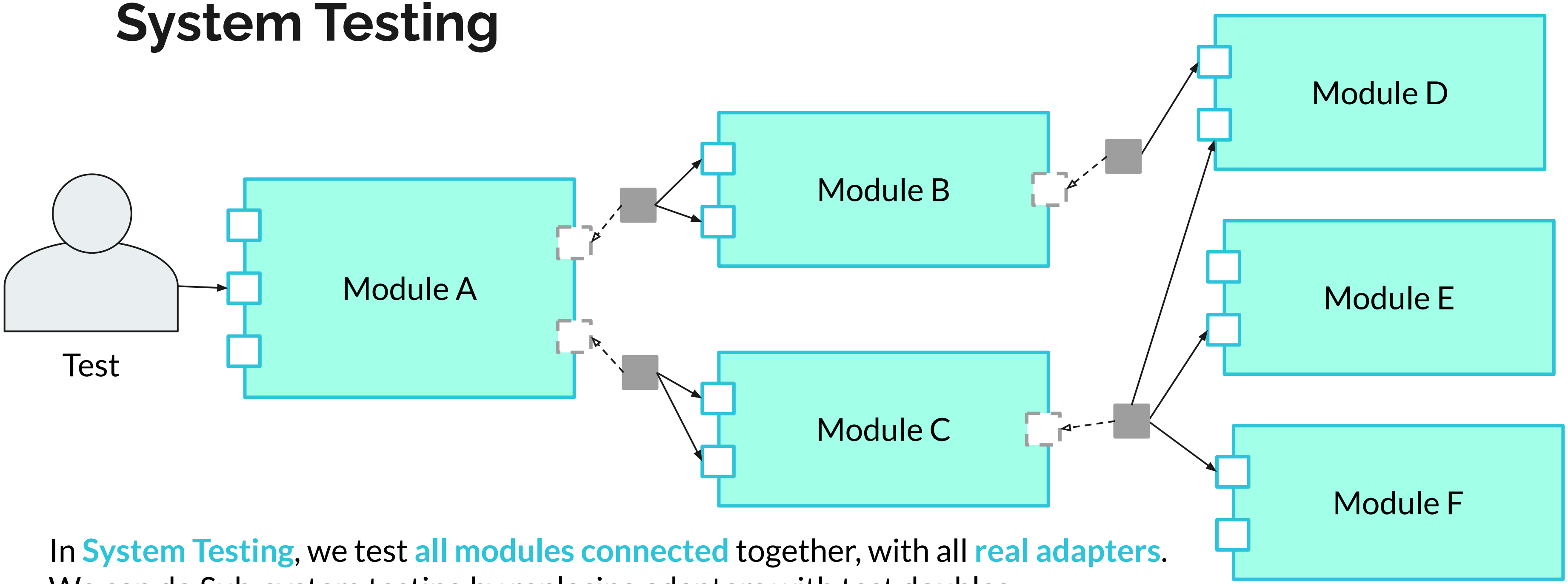
We can safely refactor the module implementation without breaking the tests (robust tests).

Module Dependencies & Test Doubles



A **module depends on abstractions** of other modules, not their implementations. The abstractions may be implemented by a **real adapter** or a **test double**. We can **test each module in isolation**, because other modules are replaced by test doubles.

System Testing



In **System Testing**, we test **all modules connected** together, with all **real adapters**.
We can do Sub-system testing by replacing adapters with test doubles.

Photo by Adrian Celaya on Unspla

3. Clean Architecture

Hexagonal Architecture & Use Case Driven Development (UCDD)

Hexagonal Architecture

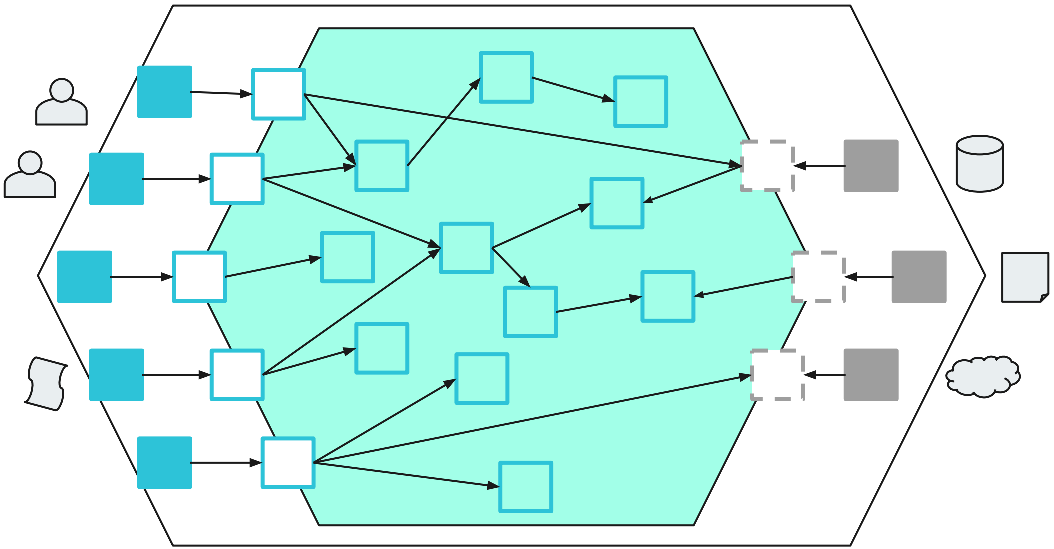
Adapted from https://alistair.cockburn.us/hexagonal-architecture/

🟩 **Application Core** (Hexagon)

👤 *Users, Programs, Scripts*

□ **User-side API** (Driver Ports)

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



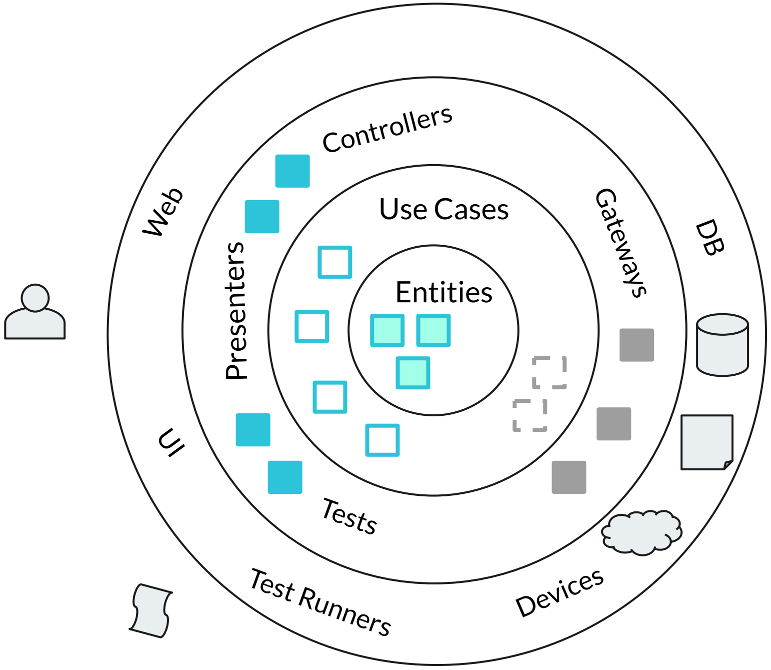
⬚ **Server-side API** (Driven Ports)

■ **Server-side Adapters** (Driven Adapters)

🛢 *DB, Files, Web Services*

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

Clean Architecture & Hexagonal Architecture

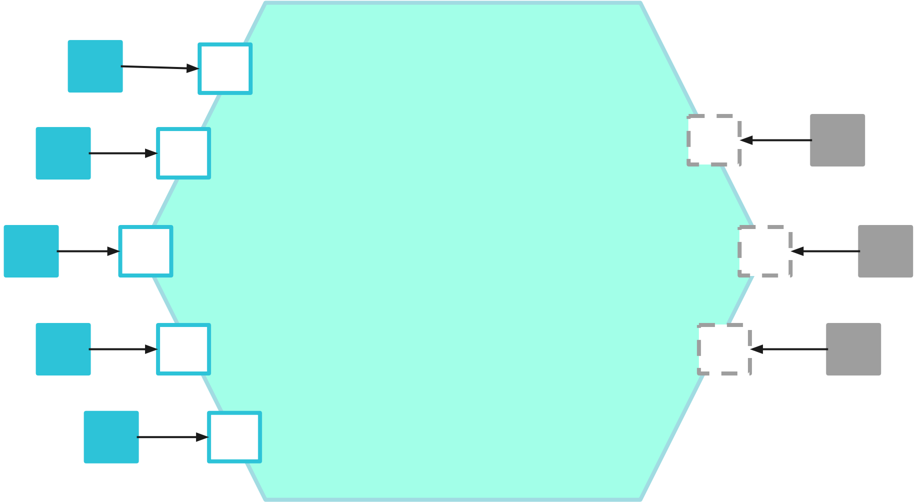


Hexagonal Architecture

- 👤 Users
- ■ User-side Adapters
- □ User-side Ports
- ▨ Application Core
- ⬚ Server-side Ports
- ■ Server-side Adapters
- ⛁ Technologies

Unit Testing & Hexagonal / Clean Architecture

Unit Tests interact with the system through the user-side API (use cases)



Test Doubles serve as in-memory adapters for the server-side API (gateway interfaces)

Acceptance Testing - Tests acting as the Users

Acceptance Testing - Unit Level

Unit Tests execute Use Cases

Gateway Interfaces are substituted with Test Doubles

Acceptance Testing - E2E Level

UI Automation runners execute Use Cases

Gateway Interfaces are substituted with real adapter implementations

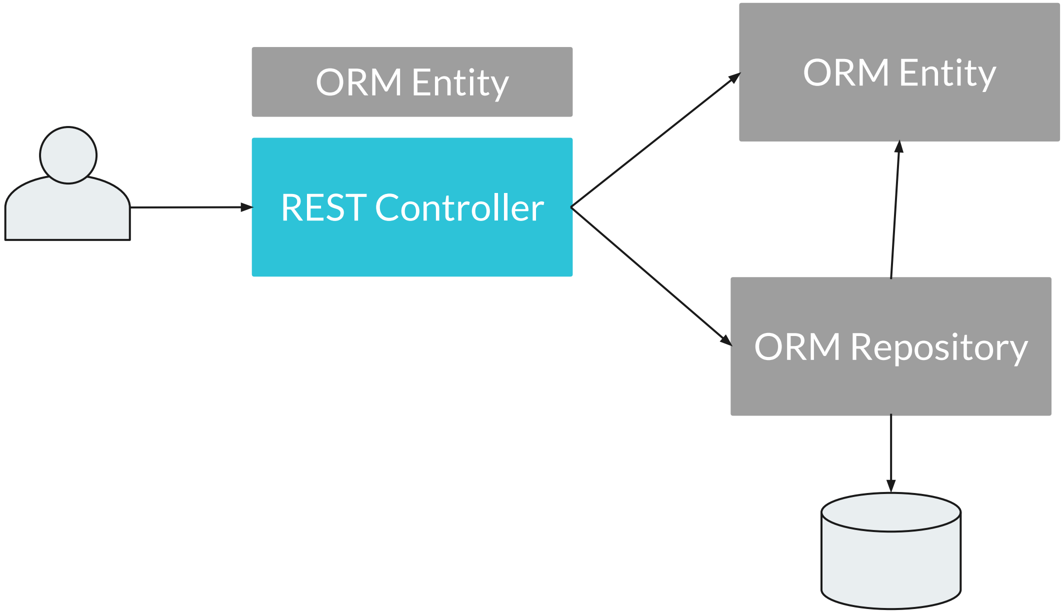
Benefit: we can run acceptance tests at the unit level through the use case ports, like the user!
Much faster feedback & scenario coverage at the unit level

Photo by Anders Jildén on Unsplash

4. Architecture Implementation

CRUD, Onion Architecture and Clean Architecture

CRUD Architecture

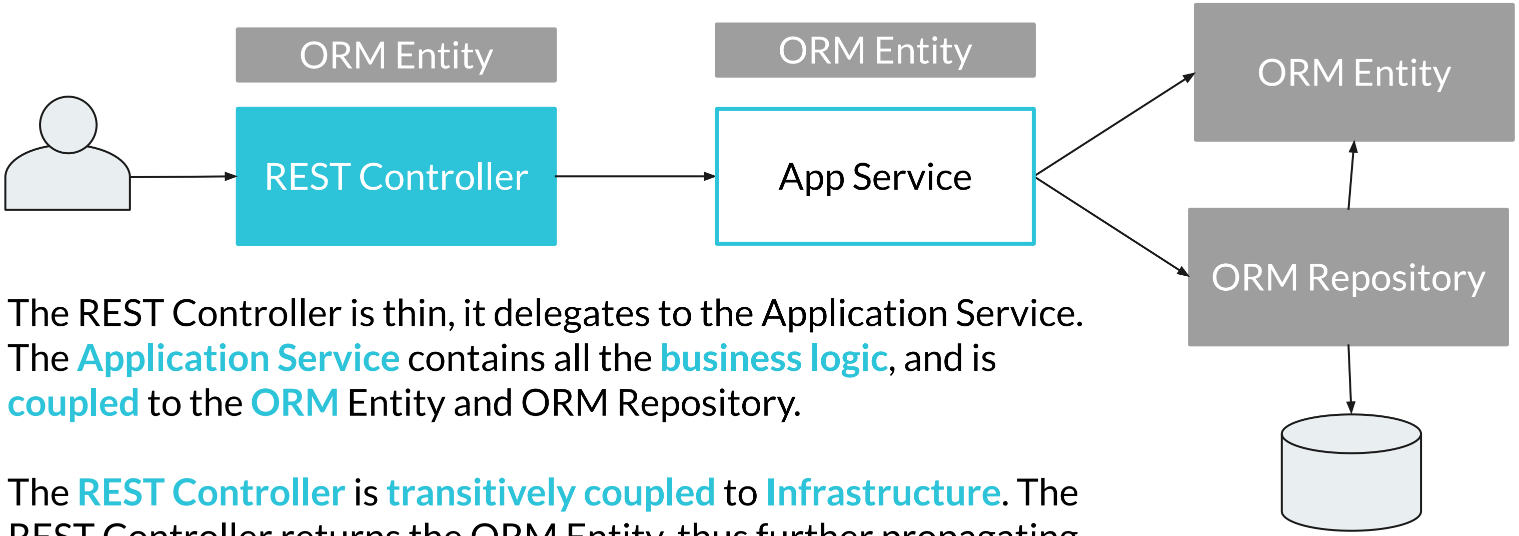


The REST Controller is directly coupled to ORM - ORM Entities & ORM Repositories.

Thus, the presentation layer is directly coupled to the infrastructure layer.

Note: "ORM Repositories" is an oxymoron, since a Repository is an abstraction over the persistence mechanism, and thus should not be coupled to some ORM. However, I used this term due to widespread usage of the term JPA "repositories", even though this is a contraction, as a repository should be framework-agnostic.

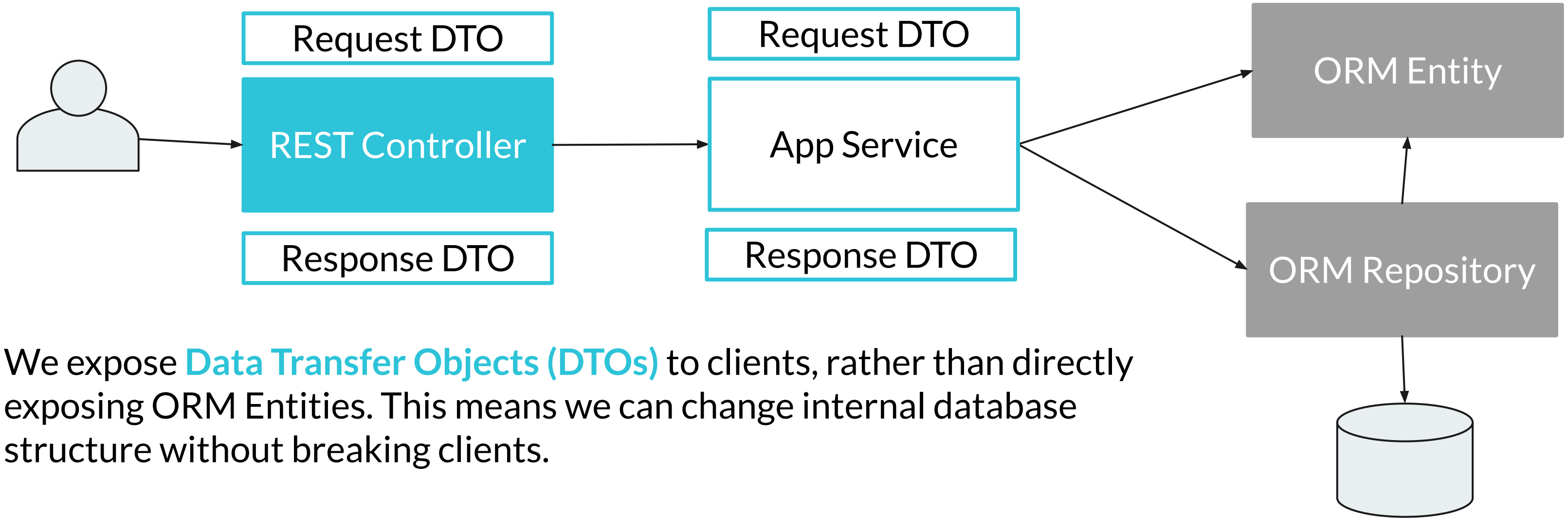
CRUD Architecture with Services



The REST Controller is thin, it delegates to the Application Service. The **Application Service** contains all the **business logic**, and is **coupled** to the **ORM** Entity and ORM Repository.

The **REST Controller** is **transitively coupled** to **Infrastructure**. The REST Controller returns the ORM Entity, thus further propagating the dependency to the client!

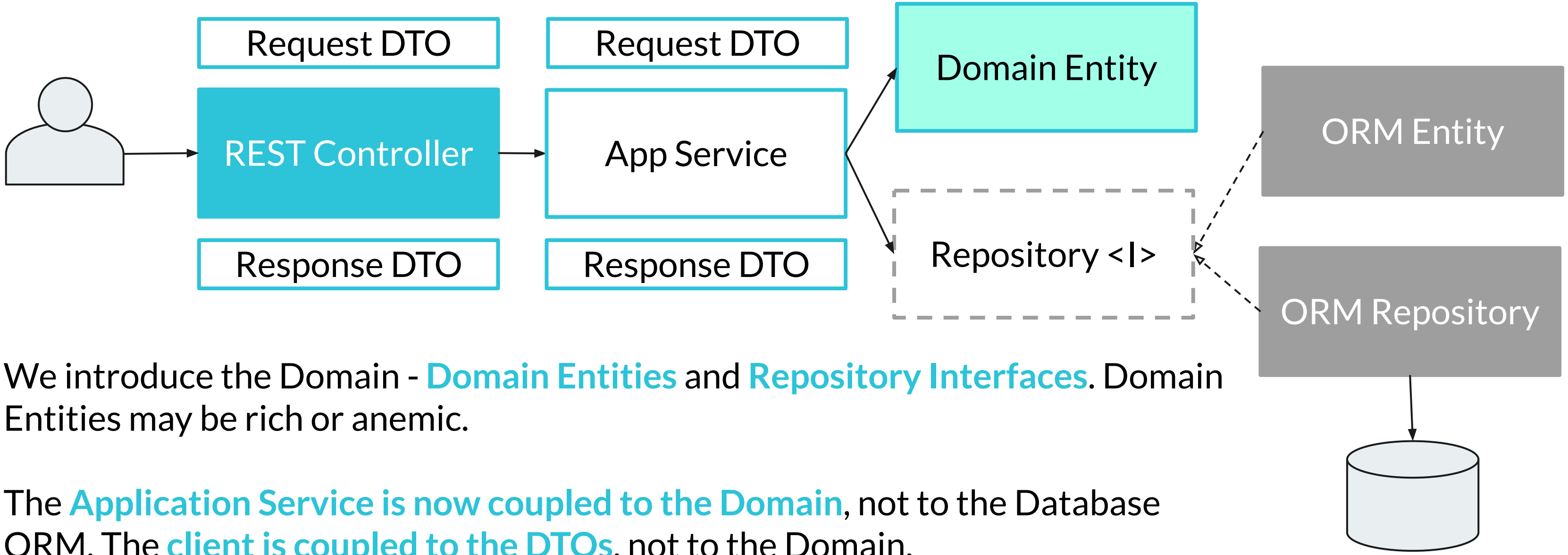
CRUD Architecture with Services and DTOs



We expose **Data Transfer Objects (DTOs)** to clients, rather than directly exposing ORM Entities. This means we can change internal database structure without breaking clients.

For better maintainability, there should be **separate DTOs per endpoint** (rather than sharing DTOs between endpoints).

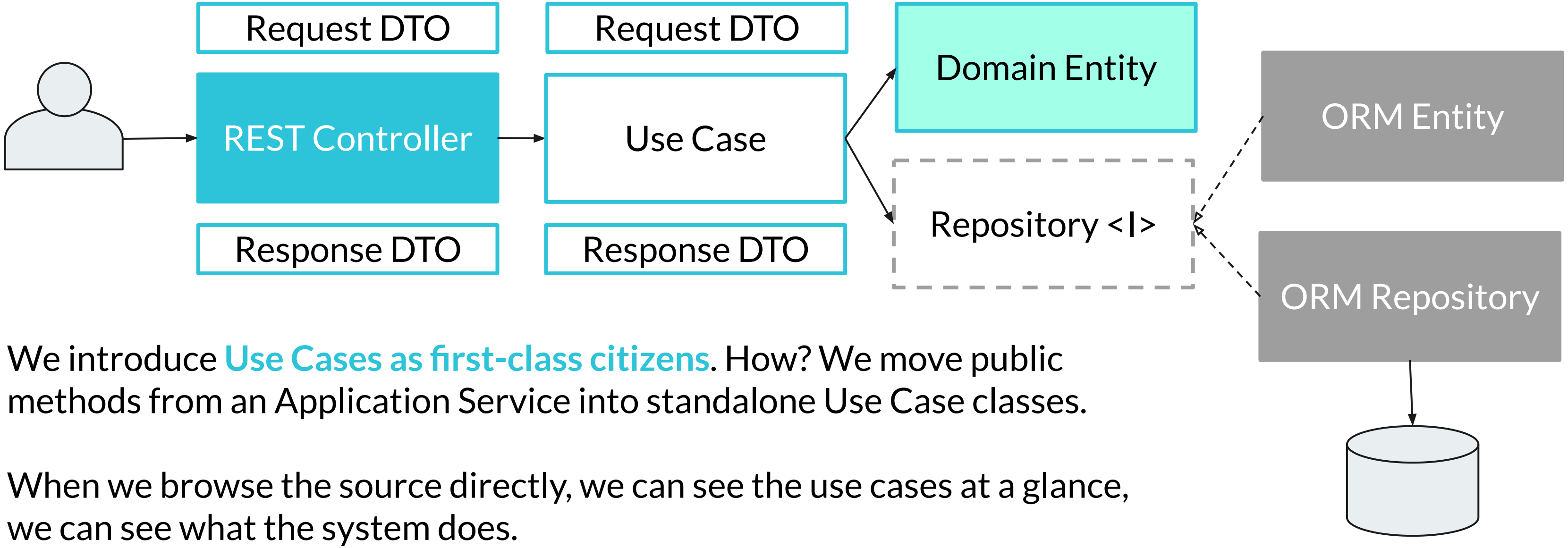
Hexagonal Architecture - Onion Architecture



We introduce the Domain - **Domain Entities** and **Repository Interfaces**. Domain Entities may be rich or anemic.

The **Application Service is now coupled to the Domain**, not to the Database ORM. The **client is coupled to the DTOs**, not to the Domain.

Hexagonal Architecture - Clean Architecture



We introduce **Use Cases as first-class citizens**. How? We move public methods from an Application Service into standalone Use Case classes.

When we browse the source directly, we can see the use cases at a glance, we can see what the system does.

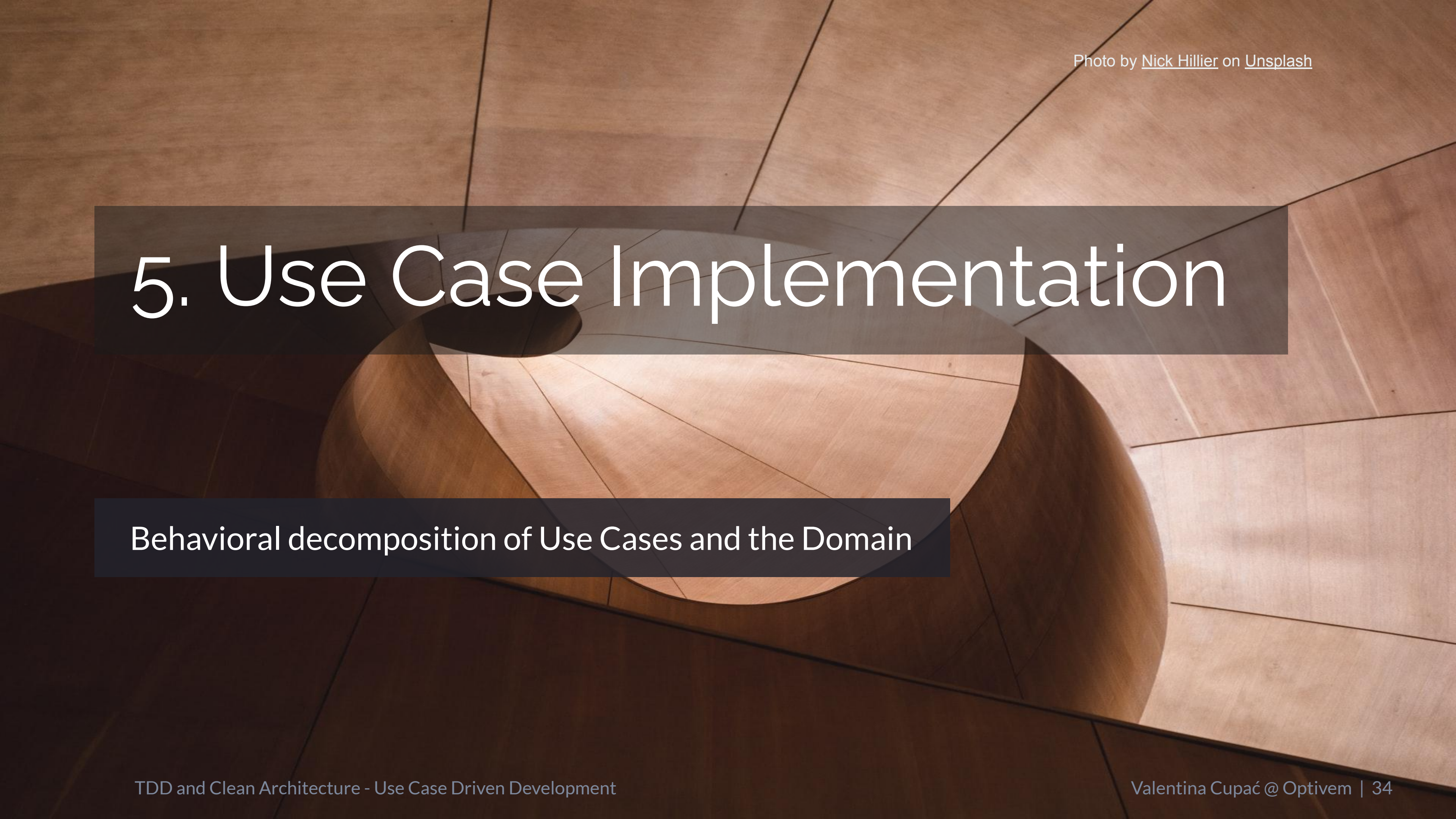
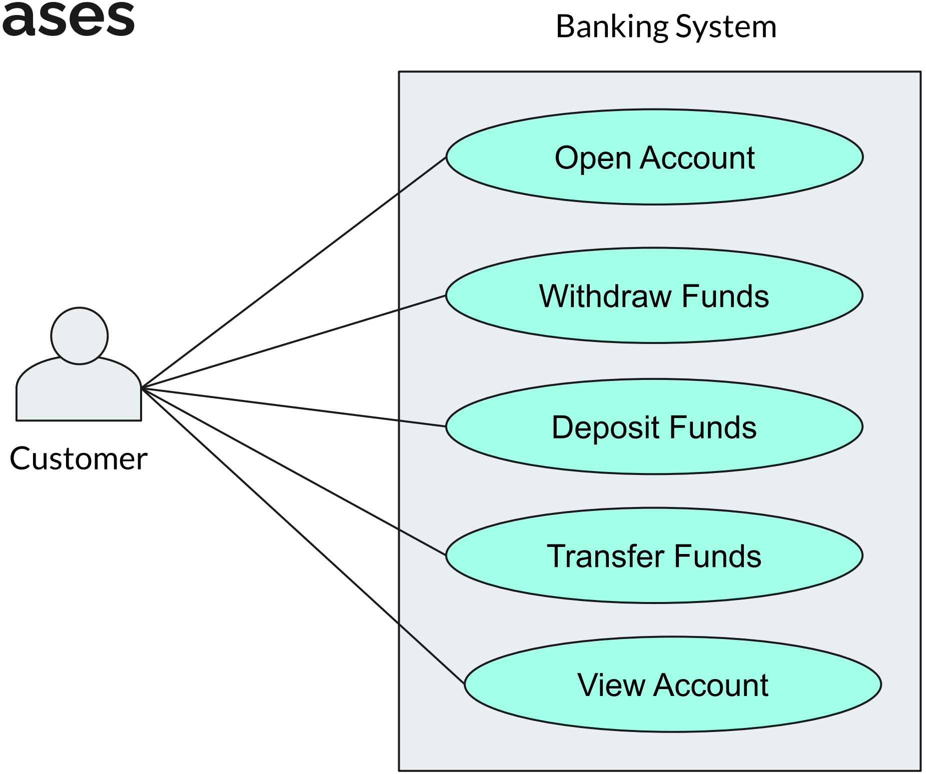


Photo by Nick Hillier on Unsplash

5. Use Case Implementation

Behavioral decomposition of Use Cases and the Domain

Use Cases



Use Cases help us model the interactions between Users and the System.

The System provides Use Cases to help Users satisfy their goals.

What’s the programming language inside the system? Which databases are used? Technology is an implementation detail.

Use Case - Withdraw Funds

1. Receive the Request DTO: { account number, withdrawal amount }
2. Is the account number empty? If yes, then exit with error
3. Is the withdrawal amount non-positive? If yes, then exit with error
4. Retrieve the bank account from the bank account repository
5. Is the bank account non-existent? If yes, then exit with error
6. Does the bank account have insufficient balance? If yes, exit with error
7. Calculate new balance: current balance minus withdrawal amount
8. Set the bank account balance to the new balance
9. Update the bank account in the bank account repository
10. Return the Response DTO: { new balance }

Implementation I - Rich Use Case, Anemic Entity

Use Case contains **all behavior** (i.e. ***WithdrawFundsUseCase*** validates the request, retrieves the account from the repository, calculates the new balance after withdrawal, and updates the account in the repository)

Domain Entity contains **only data, no behavior** (i.e. ***BankAccount*** entity is just a data structure with getters and setters which are mutated by the *WithdrawFundsUseCase*; or it could be a data structure with getters so that *WithdrawFundsUseCase* doesn’t mutate the *BankAccount* but instead constructs a new instance of the entity as a result of withdrawal)

Implementation II - Rich service, anemic entity

Use Case contains only workflow behavior (i.e. ***WithdrawFundsUseCase*** performs only the workflow - the application logic - communicating with repository and the *WithdrawService*; but it does not perform the business logic, i.e. it does not do any withdrawal calculations, nor mutating the *BankAccount* entity)

Domain Service contains all business logic behavior (i.e. ***WithdrawService*** checks whether withdrawal can be done, calculates the new balance, updates the balance of the *BankAccount* entity, and returns the entity back to the *WithdrawFundsUseCase*)

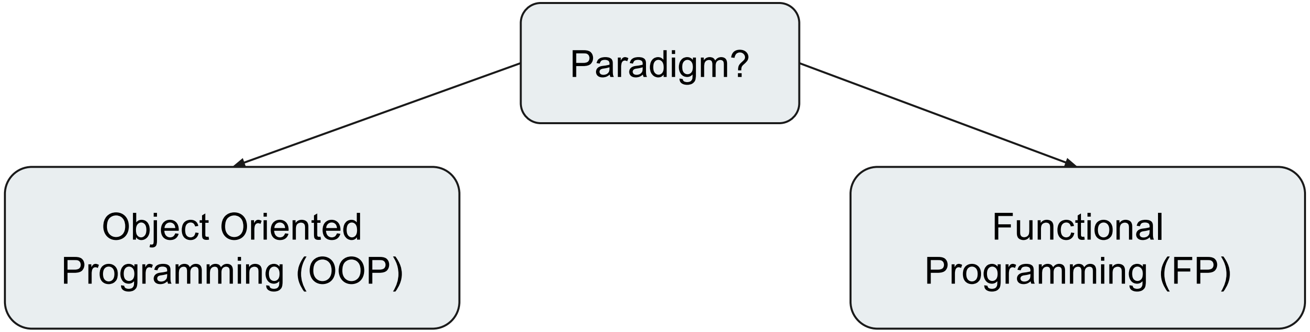
Domain Entity contains only data, no behavior (i.e. ***BankAccount*** entity is just a data structure with getters and setters which are mutated by the *WithdrawService*; or it could be a data structure with getters so that the *WithdrawService* doesn't mutate the *BankAccount* but instead constructs a new instance of the entity when there is a change)

Implementation III - Rich entity

Use Case contains **only workflow behavior** (i.e. ***WithdrawFundsUseCase*** performs only the workflow - the application logic - communicating with repository and the *BankAccount*; but it does not perform the business logic, i.e. it does not do any withdrawal calculations)

Domain Entity contains **all business logic behavior** (i.e. *BankAccount* checks whether withdrawal can be done, calculates the balance, updates itself; alternatively the *BankAccount* may be immutable whereby the withdraw method will return a new instance of the *BankAccount* with the new balance)

A summary of implementation choices


Paradigm?
Object Oriented Programming (OOP)
Functional Programming (FP)

Most behavior should be in Rich Domain Entities (stateful, mutable). Rarely in Domain Services.

Most behavior should be in Domain Services (functions) working with immutable data structures.

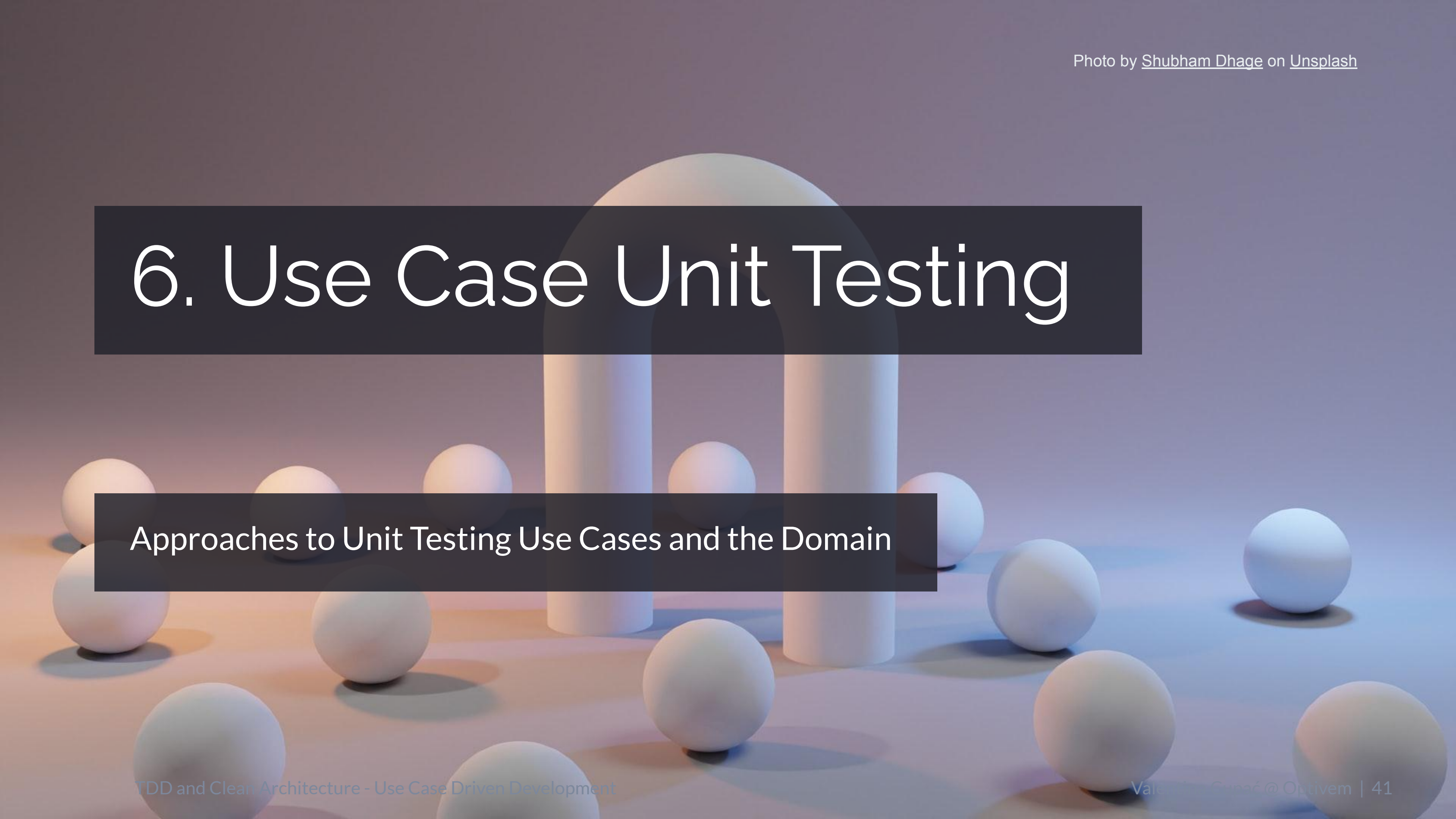
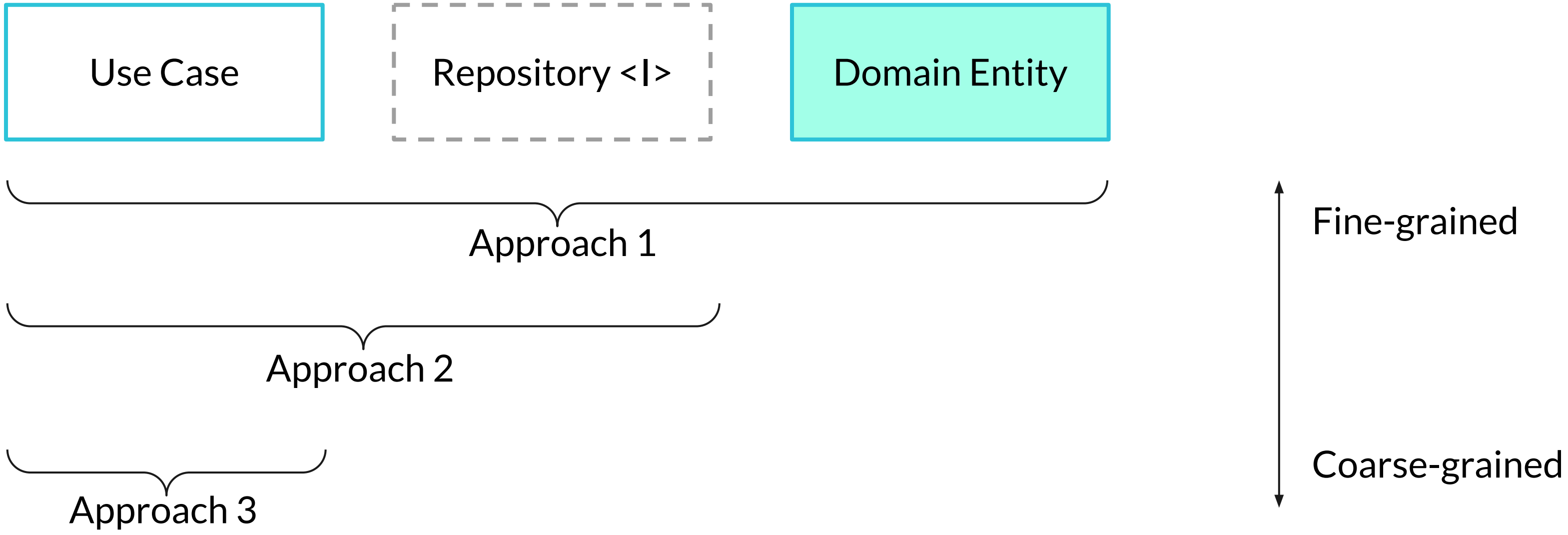


Photo by Shubham Dhage on Unsplash

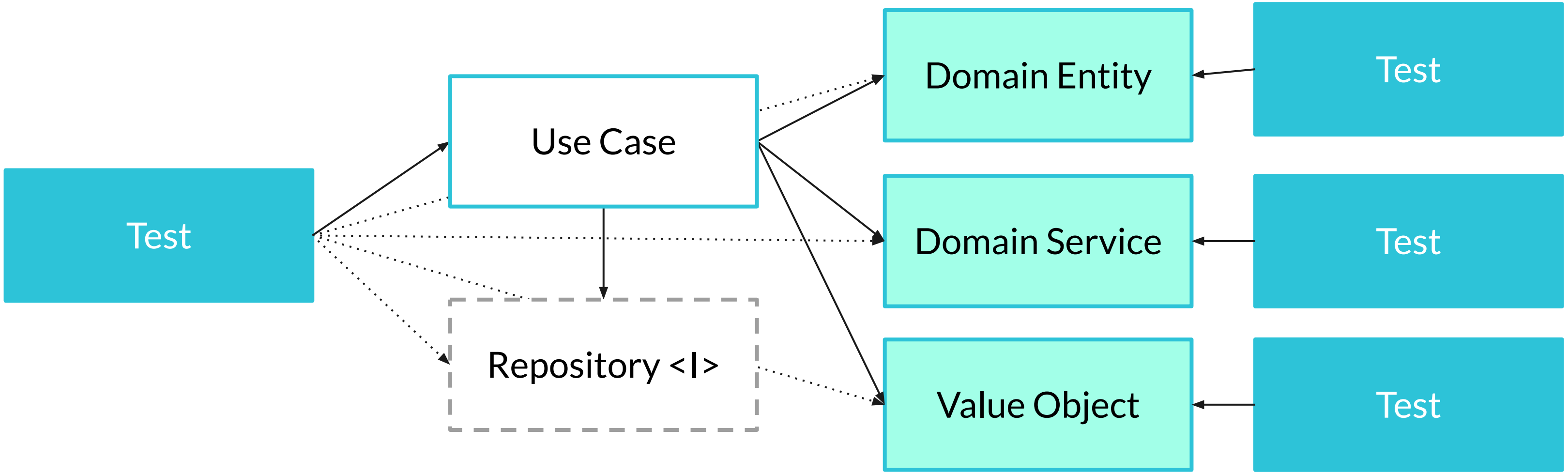
6. Use Case Unit Testing

Approaches to Unit Testing Use Cases and the Domain

Unit Testing Use Cases - Overview of Approaches

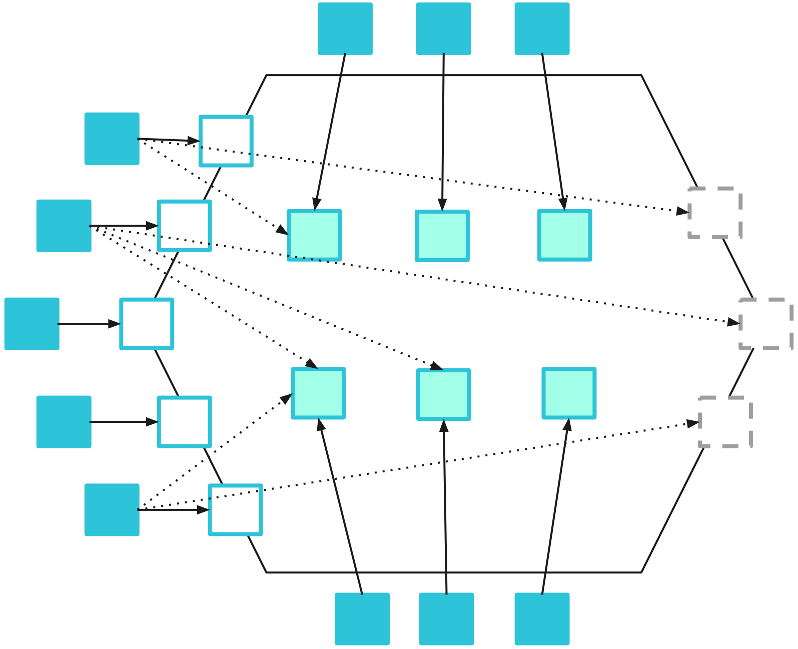


Approach 1 - Testing Use Cases, Gateways & Domain



Testing **behavioral decomposition** in **isolation**. We test application logic through Use Cases, testing business logic through the Domain. We assert side-effects on Gateways using Test Doubles.

Approach 1 - Testing Use Cases, Gateways & Domain



- ■ Tests
- □ Use Cases
- 🟩 Domain
- ⬚ Gateway Interfaces

Approach 1 - Testing Use Cases, Gateways & Domain

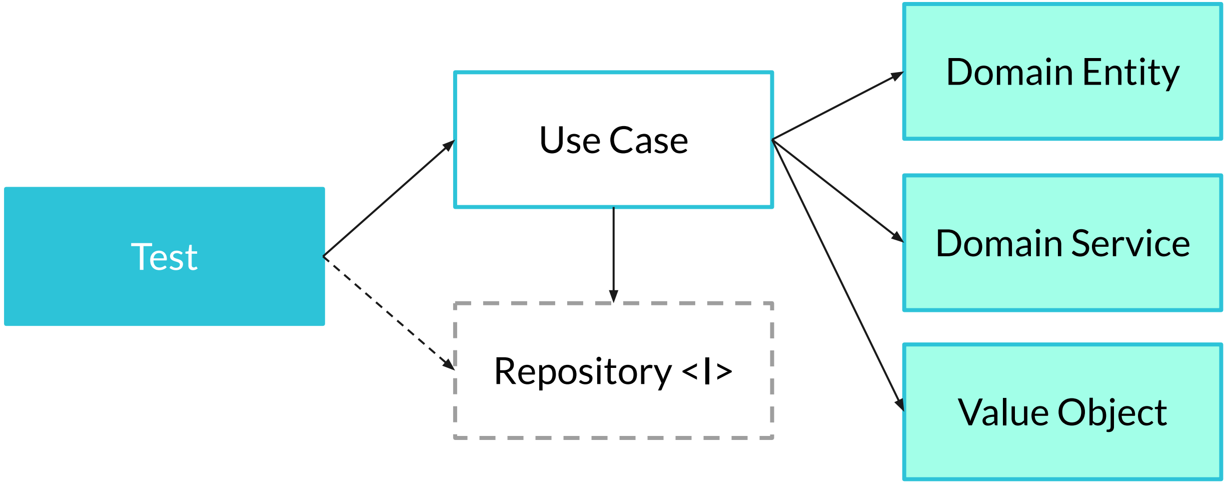
Advantages

- Testing behavioral decomposition at the lowest level of granularity
- Useful for handling combinatorial explosion within the domain
- Useful for handling mathematically complex domains by testing interim steps

Disadvantages

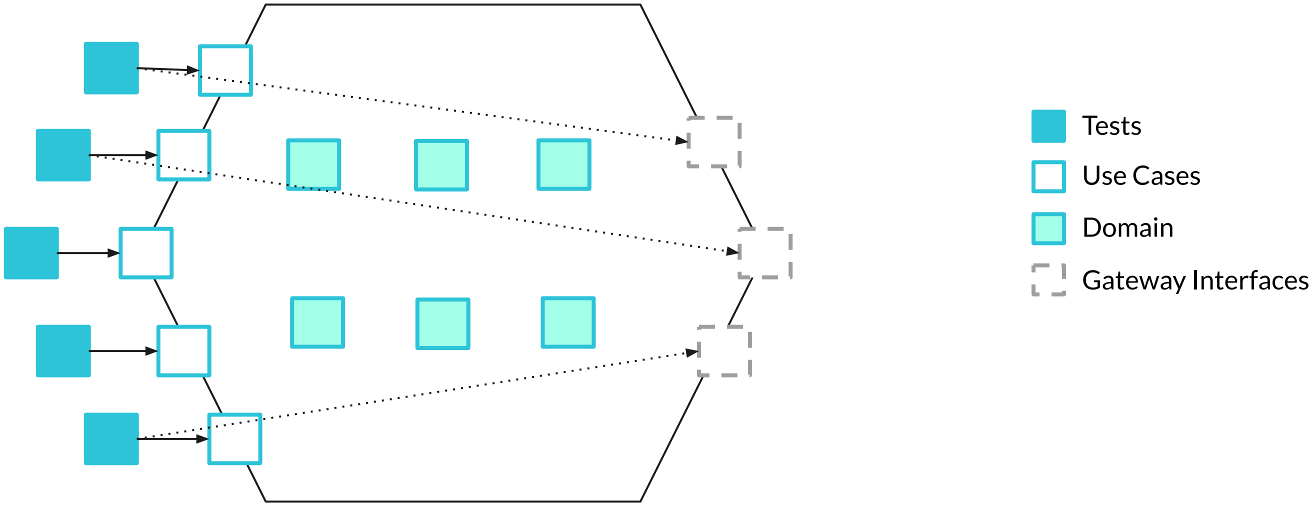
- We lose oversight of the use case requirements, need to look at multiple test files
- More test code, more expensive to write these tests and more expensive to maintain
- Tests are coupled to system implementation, sensitive to changes in behavioral decomposition and usage of design patterns, thus they are fragile and hinder refactoring

Approach 2 - Testing Use Cases & Gateways



We test the **Use Case behavior** without being coupled to the implementation (behavioral decomposition), the Domain is transitively covered. We assert side-effects on Gateways using Test Doubles.

Approach 2 - Testing Use Cases & Gateways



- ■ Tests
- □ Use Cases
- ▨ Domain
- ⬚ Gateway Interfaces

Approach 2 - Testing Use Cases & Gateways

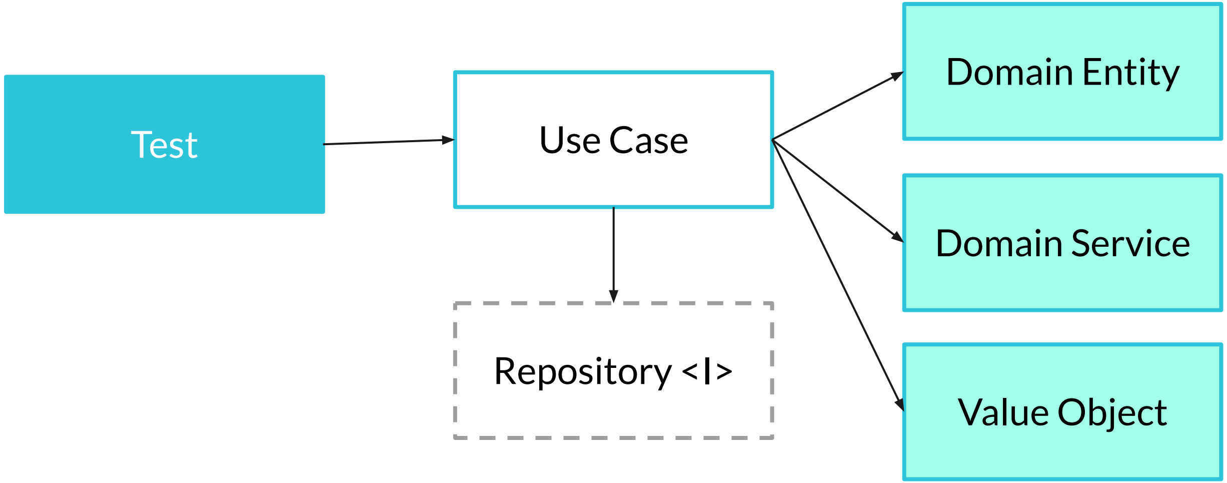
Advantages

- Targeting the API of the system (Use Cases & Gateways), not the implementation (Domain)
- Tests are readable as requirement specifications (main flows, exceptional flows)
- High coverage at a low cost - less test code, less expensive to write & maintain tests
- We can refactor system implementation, we can iteratively discover the domain and change behavioral decomposition without breaking the tests - high test robustness

Disadvantages

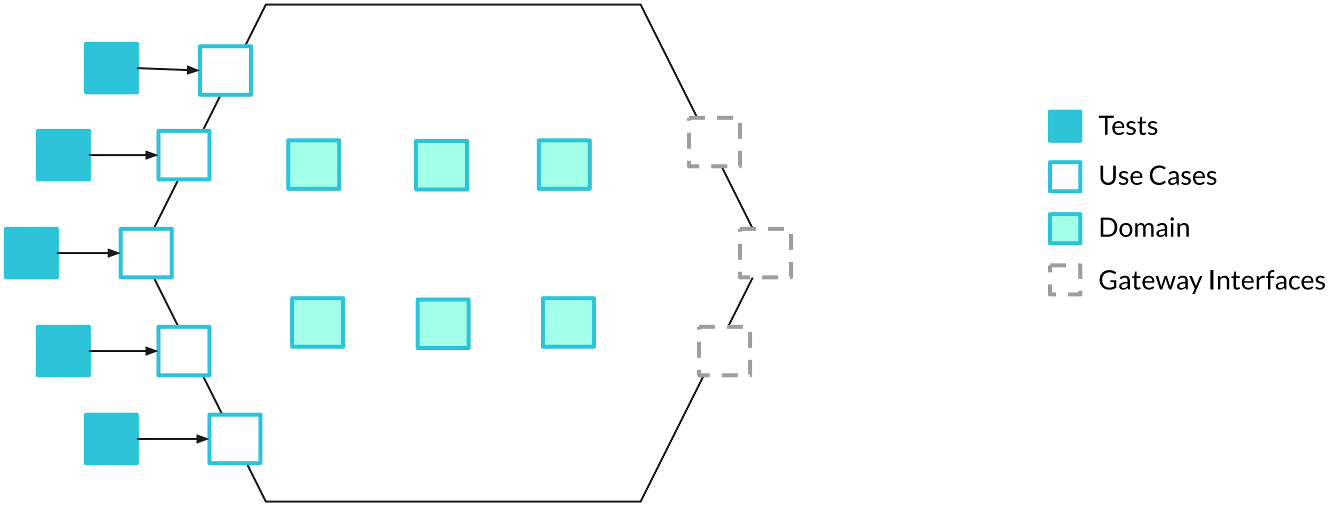
- For use cases with behavioral combinatorial explosion and/or mathematical complexity, we may need to supplement use case tests with lower-level granular tests (“shifting gears”)

Approach 3 - Testing Use Cases



Our test is targeting **only Use Cases**, the Domain is transitively covered. We do not assert side-effects on the Gateways, but rather **query the system state by executing other Use Cases**.

Approach 3 - Testing Use Cases



Approach 3 - Testing Use Cases

Advantages

- Targeting the API of the system (Use Cases), not the implementation (Domain)
- Tests are readable as requirement specifications (main flows, exceptional flows)
- High coverage at a low cost - less test code, less expensive to write & maintain tests
- Highest test robustness because the tests are coupled only to Use Cases, nothing else

Disadvantages

- Unable to verify side-effects on the Gateways in cases where such side-effects are not visible through Use Cases *(e.g. the system sets some internal numbers or timestamps on entities updated to the repository, but not exposed through any query Use Cases)*

Which approach is optimal?

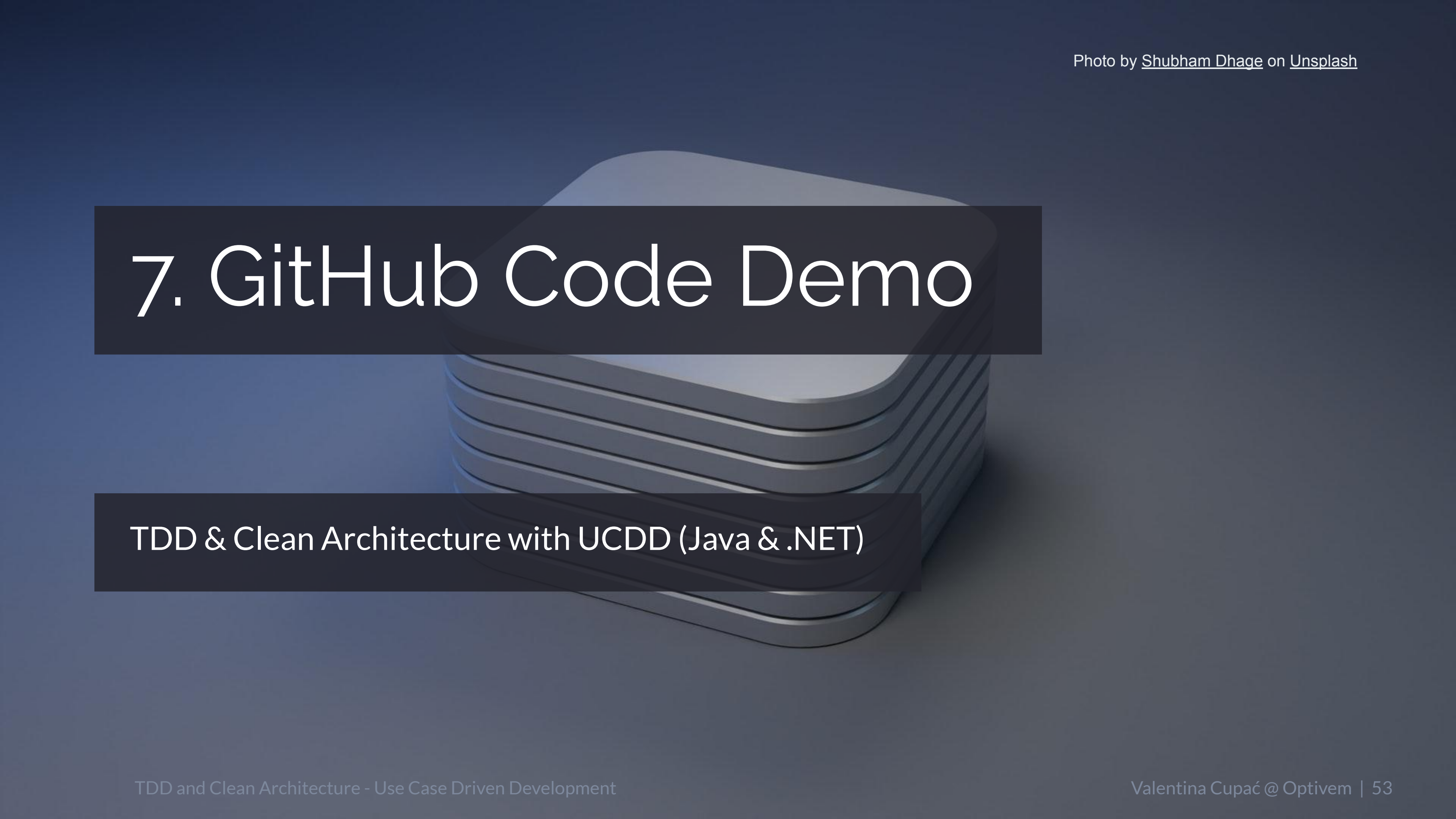
Prefer testing Use Cases (coarse-grained testing), transitively covering the Domain, because that provides **highest ROI**.

We generally adopt **unit testing Use Cases and Gateways** (Approach 2) because:

- Use Case tests are **coupled to the system API**, not the system implementation
- Use Case tests are **readable as requirement specs**, rather than implementation specs
- **High coverage** at **low test maintenance costs**, thus increasing ROI
- **High test robustness**, we can **refactor the system safely** without breaking tests

*For some Use Cases where we see high **combinatorial explosion** and/or algorithmic complexity whereby we need to test interim behavioral steps, we then supplement the Use Case tests with **lower-level fine-grained tests**. We have a trade-off, here we are prepared to have higher cost of writing tests, test fragility because the tests are coupled to system implementation - but we do it so that these tests can help us with implementation.*

Photo by Shubham Dhage on Unsplash



7. GitHub Code Demo

TDD & Clean Architecture with UCDD (Java & .NET)

GitHub Code Demo

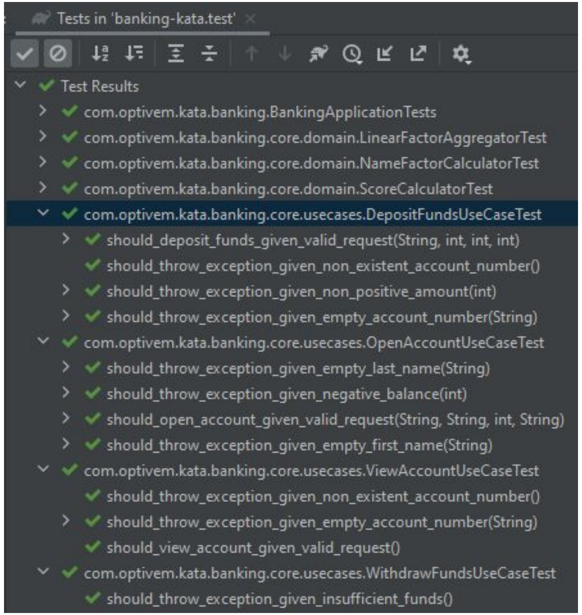
The following open source GitHub projects illustrate **TDD & Clean Architecture** with a **Use Case Driven Development** (UCDD) approach. They show an incremental and iterative approach to implementing use cases with a robust test suite.

Banking Kata (Java) https://github.com/valentinacupac/banking-kata-java

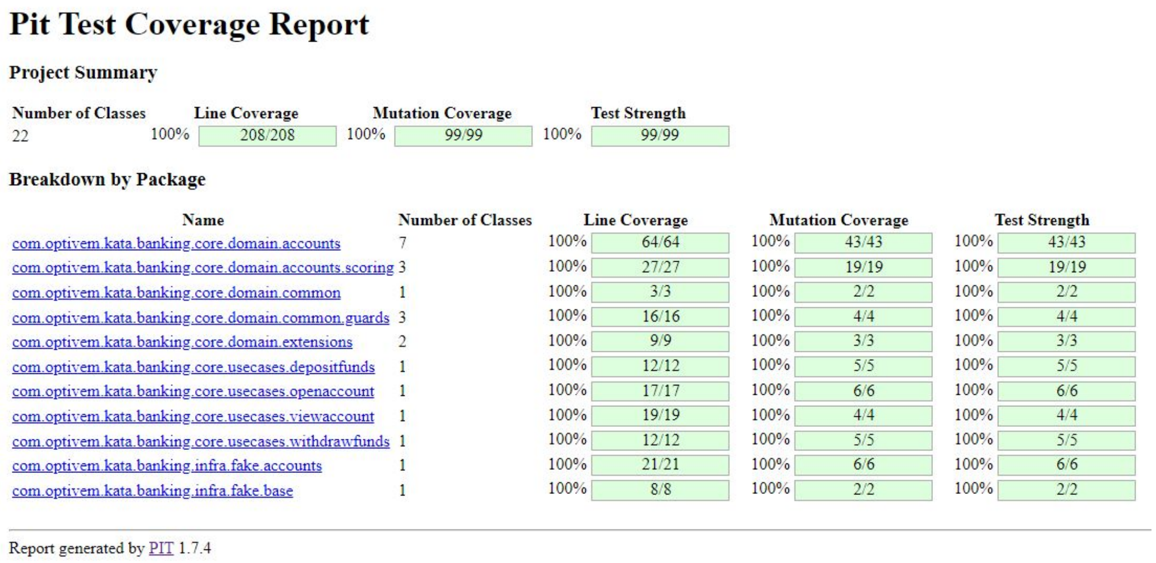
Banking Kata (.NET) https://github.com/valentinacupac/banking-kata-dotnet

I am continuing development on these projects; you can **follow me on GitHub** https://github.com/valentinacupac to get further updates. *Feel free to contact me if you have any questions, feedback or suggestions regarding these demo projects.*

Java - Unit Tests & Mutation Testing



```
✓ Test Results
  ✓ com.optivem.kata.banking.BankingApplicationTests
  ✓ com.optivem.kata.banking.core.domain.LinearFactorAggregatorTest
  ✓ com.optivem.kata.banking.core.domain.NameFactorCalculatorTest
  ✓ com.optivem.kata.banking.core.domain.ScoreCalculatorTest
  ✓ com.optivem.kata.banking.core.usecases.DepositFundsUseCaseTest
    ✓ should_deposit_funds_given_valid_request(String, int, int, int)
    ✓ should_throw_exception_given_non_existent_account_number()
    ✓ should_throw_exception_given_non_positive_amount(int)
    ✓ should_throw_exception_given_empty_account_number(String)
  ✓ com.optivem.kata.banking.core.usecases.OpenAccountUseCaseTest
    ✓ should_throw_exception_given_empty_last_name(String)
    ✓ should_throw_exception_given_negative_balance(int)
    ✓ should_open_account_given_valid_request(String, String, int, String)
    ✓ should_throw_exception_given_empty_first_name(String)
  ✓ com.optivem.kata.banking.core.usecases.ViewAccountUseCaseTest
    ✓ should_throw_exception_given_non_existent_account_number()
    ✓ should_throw_exception_given_empty_account_number(String)
    ✓ should_view_account_given_valid_request()
  ✓ com.optivem.kata.banking.core.usecases.WithdrawFundsUseCaseTest
    ✓ should_throw_exception_given_insufficient_funds()
```



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
22	100%	208/208	100%	99/99	100%	99/99

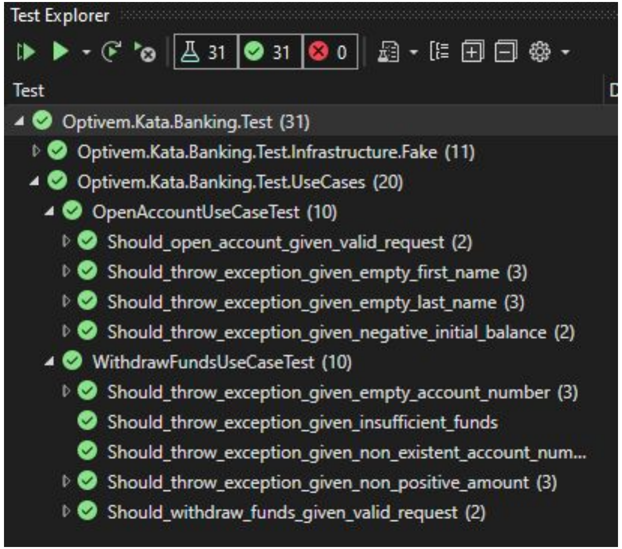
Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
com.optivem.kata.banking.core.domain.accounts	7	100%	64/64	100%	43/43	100%	43/43
com.optivem.kata.banking.core.domain.accounts.scoring	3	100%	27/27	100%	19/19	100%	19/19
com.optivem.kata.banking.core.domain.common	1	100%	3/3	100%	2/2	100%	2/2
com.optivem.kata.banking.core.domain.common.guards	3	100%	16/16	100%	4/4	100%	4/4
com.optivem.kata.banking.core.domain.extensions	2	100%	9/9	100%	3/3	100%	3/3
com.optivem.kata.banking.core.usecases.depositfunds	1	100%	12/12	100%	5/5	100%	5/5
com.optivem.kata.banking.core.usecases.openaccount	1	100%	17/17	100%	6/6	100%	6/6
com.optivem.kata.banking.core.usecases.viewaccount	1	100%	19/19	100%	4/4	100%	4/4
com.optivem.kata.banking.core.usecases.withdrawfunds	1	100%	12/12	100%	5/5	100%	5/5
com.optivem.kata.banking.infra.fake.accounts	1	100%	21/21	100%	6/6	100%	6/6
com.optivem.kata.banking.infra.fake.base	1	100%	8/8	100%	2/2	100%	2/2

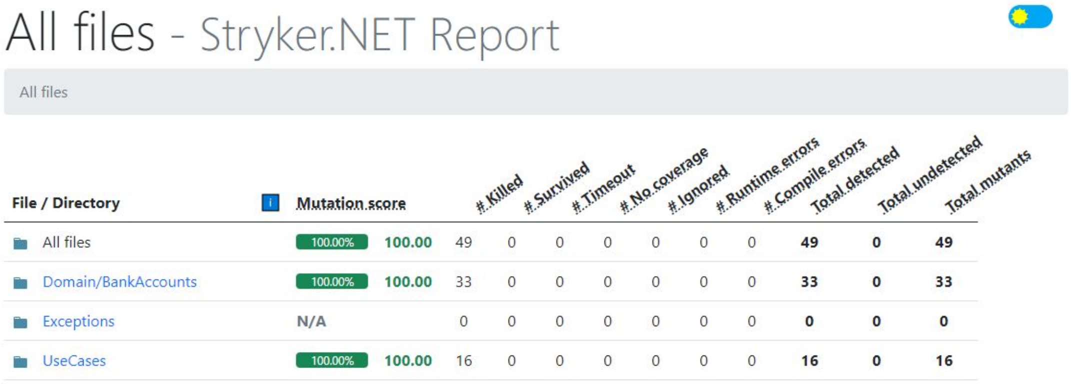
Report generated by PIT 1.7.4

See instructions in the README.md file in https://github.com/valentinacupac/banking-kata-java

.NET Unit Tests & Mutation Testing



```
◢ Optivem.Kata.Banking.Test (31)
  ▷ Optivem.Kata.Banking.Test.Infrastructure.Fake (11)
  ◢ Optivem.Kata.Banking.Test.UseCases (20)
    ◢ OpenAccountUseCaseTest (10)
      ▷ Should_open_account_given_valid_request (2)
      ▷ Should_throw_exception_given_empty_first_name (3)
      ▷ Should_throw_exception_given_empty_last_name (3)
      ▷ Should_throw_exception_given_negative_initial_balance (2)
    ◢ WithdrawFundsUseCaseTest (10)
      ▷ Should_throw_exception_given_empty_account_number (3)
        Should_throw_exception_given_insufficient_funds
        Should_throw_exception_given_non_existent_account_num...
      ▷ Should_throw_exception_given_non_positive_amount (3)
      ▷ Should_withdraw_funds_given_valid_request (2)
```



All files - Stryker.NET Report

All files

File / Directory	Mutation score		# Killed	# Survived	# Timeout	# No coverage	# Ignored	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
All files	100.00%	100.00	49	0	0	0	0	0	0	**49**	**0**	**49**
Domain/BankAccounts	100.00%	100.00	33	0	0	0	0	0	0	**33**	**0**	**33**
Exceptions	N/A		0	0	0	0	0	0	0	**0**	**0**	**0**
UseCases	100.00%	100.00	16	0	0	0	0	0	0	**16**	**0**	**16**

See instructions in the README.md file in https://github.com/valentinacupac/banking-kata-dotnet

Conclusion

Tests should be executable Requirement Specs… not Implementation Specs

Tests should be coupled to the System API… not the System Implementation

Use Case Driven Design helps us drive the system from through Use Cases

Clean Architecture exposes Use cases, encapsulating the Domain

Use Case Unit Tests are Acceptance Tests of System Behavior

Benefits are higher robustness & lower maintenance costs → higher ROI ✔

Photo by JJ Ying on Unsplash

Thank You

Valentina Cupać @ Optivem
Founder Technical Coach

E valentina.cupac@optivem.com
W www.optivem.com

Connect on: LinkedIn | Twitter | GitHub