TDD et Clean Architecture Guidé par le Comportement

Valentina Cupać - Optivem

À propos de l'orateur

Valentina Cupać entraîne des équipes de développement en TDD & Clean Architecture pour augmenter la qualité, accélérer la livraison et faire évoluer les équipes..

Auparavant, elle a travaillé en tant que Développeur Senior, Responsable Technique et Architecte de Solutions.

Diplômée de l'Université de Sydney - Informatique, Mathématiques et Finance.

J'écris régulièrement des articles sur LinkedIn sur le TDD & la Clean Architecture.

Connectez-vous avec moi ou **suivez-moi** sur LinkedIn:

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

Sommaire

Comment En Sommes-Sous Arrivés Là?

Le TDD est douloureux, mais est-ce une fatalité?

Idée reçue #1 - La classe est l'unité d'isolement

Il faut écrire une **classe de test** pour chaque **classe de production**.

Il faut écrire **plusieurs méthodes de test** pour chaque **méthode de production**.

Il faut isoler la classe testée en **mockant** tous ses collaborateurs.

Wikipédia dit que le test unitaire signifie tester des "**unités individuelles de code source**", et dans le cas de la POO nous testons une "**classe**, ou une **méthode** individuelle". Nous faisons confiance à Wikipédia... n'est-ce pas?

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

Idée reçue #2 - Les tests unitaires doivent être coûteux

Il est normal que le **code de test** soit **2 à 4 fois plus grand** que le code de production.

Il est normal que l'écriture de tests unitaires **prenne beaucoup de temps**.

Il est normal que les **tests unitaires cassent** lorsque l'on **refactor** la conception interne de la classe

Tout ce qui en vaut la peine doit être douloureux. *No pain, no gain*, non?

Idée reçue #3: BDD concerne le comportement, mais pas le TDD

TDD et BDD concernent les comportements. Il s'agit de tester notre système du point de vue de l'utilisateur.

TDD ne concerne pas le comportement du système, il s'agit de **tester les classes** et leurs interactions avec d'autres classes.

Lorsque nous sommes sous pression et que le budget est serré, **gardons simplement ATDD/BDD**. Il nous indique en fait si nous avons satisfait les besoins des utilisateurs.

Mais si nous pouvions résoudre les douleurs du TDD?

Imaginez si TDD pouvait vraiment **accélérer le développement**?

Imaginez si TDD pouvait être fait avec **beaucoup moins de code de test**?

Imaginez **si les tests ne cassaient pas tout le temps** pendant que vous refactoriez vos conceptions de classe?

Imaginez si vous pouviez **tester les exigences au niveau de l'unité** et obtenir des commentaires très rapides?

Imaginez si tout le monde - et **pas seulement les entreprises aux budgets énormes** - pouvaient bénéficier des avantages de TDD?

La raison sous-jacente

Notre travail n'est pas de livrer du code, notre travail est de résoudre les besoins de l'entreprise

Pourquoi

Pourquoi construisons-nous des maisons? **Pour** avoir un endroit où vivre.

Pourquoi construisons-nous des voitures? **Pour** pouvoir voyager.

Pourquoi construisons-nous des logiciels? **Pour** satisfaire les besoins des utilisateurs.

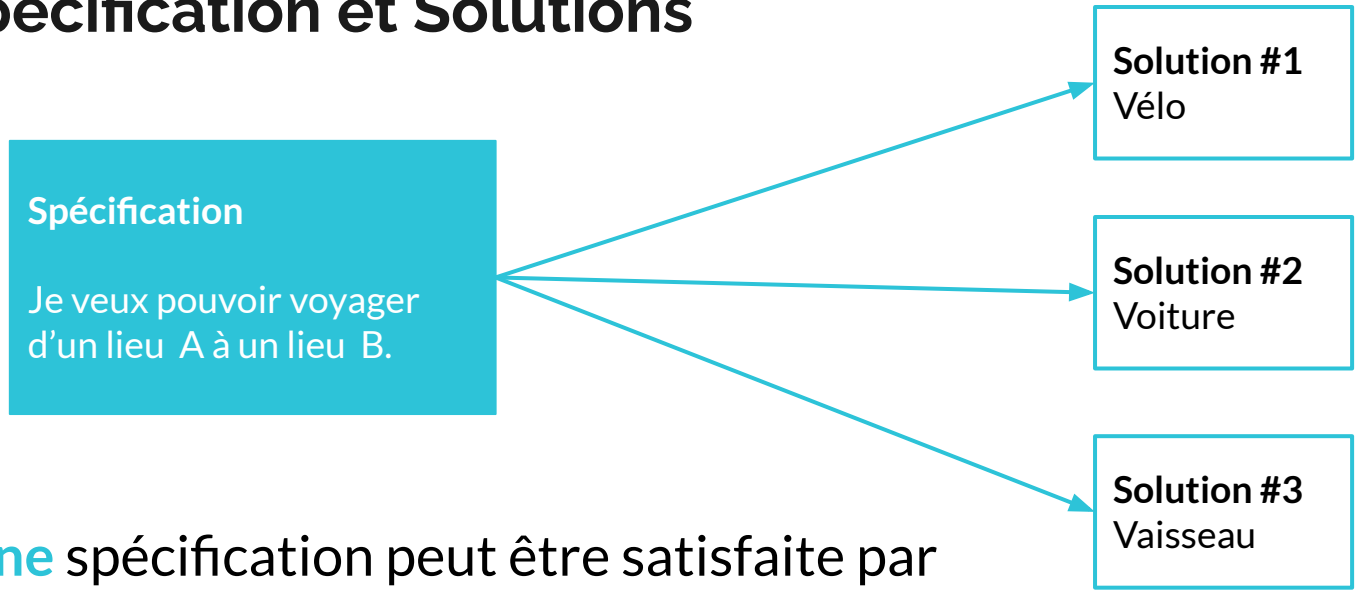
Pourquoi

Nous ne sommes pas payés pour “écrire du code”.

Nous sommes payés pour résoudre les **besoins** des utilisateurs.

Comment ? En convertissant les **spécifications** en **solutions** logicielles pour résoudre les besoins de l'entreprise.

Spécification et Solutions



Une spécification peut être satisfaite par **plusieurs** solutions

Tests En Tant Que Spécifications Exécutables

Les tests codifient-ils les spécifications fonctionnelles ou la bonne implémentation technique?

Sondage d'Audience

Connaissez-vous le terme “spécifications exécutables” ?

1. Oui
1. Pas du tout
2. À peu près

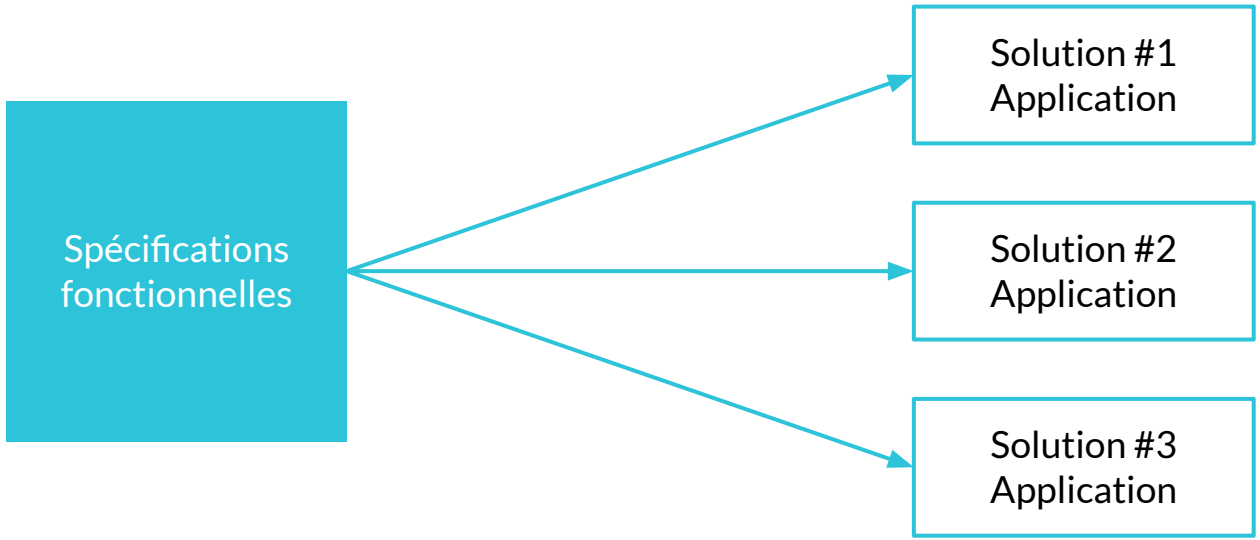
Les exigences guident l'implémentation

Besoin de satisfaire des besoins métiers, besoin de fournir une valeur métier

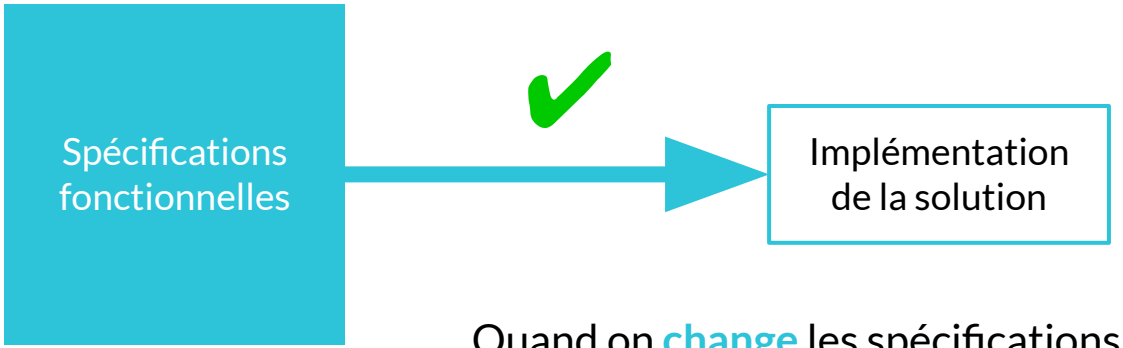
Identifier les exigences, **qui seront converties en tests**

Implémentez les exigences dans le code, **en utilisant des tests pour guider l'implémentation**

Exigences et mise en œuvre

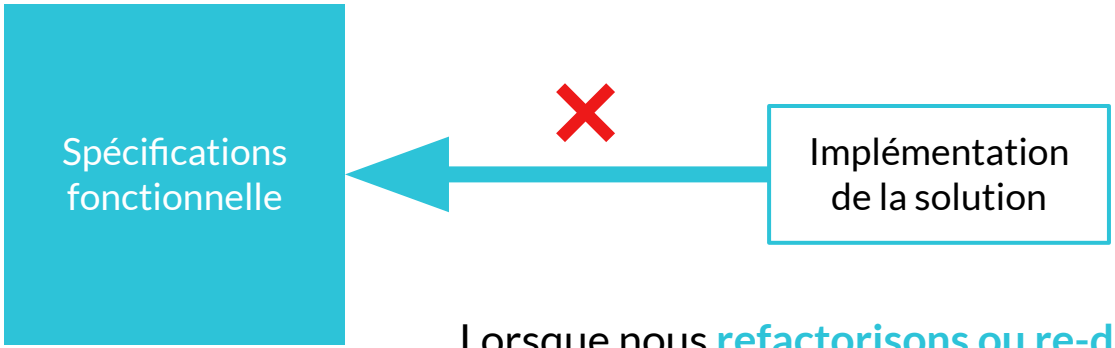


Les exigences affectent naturellement l'implémentation



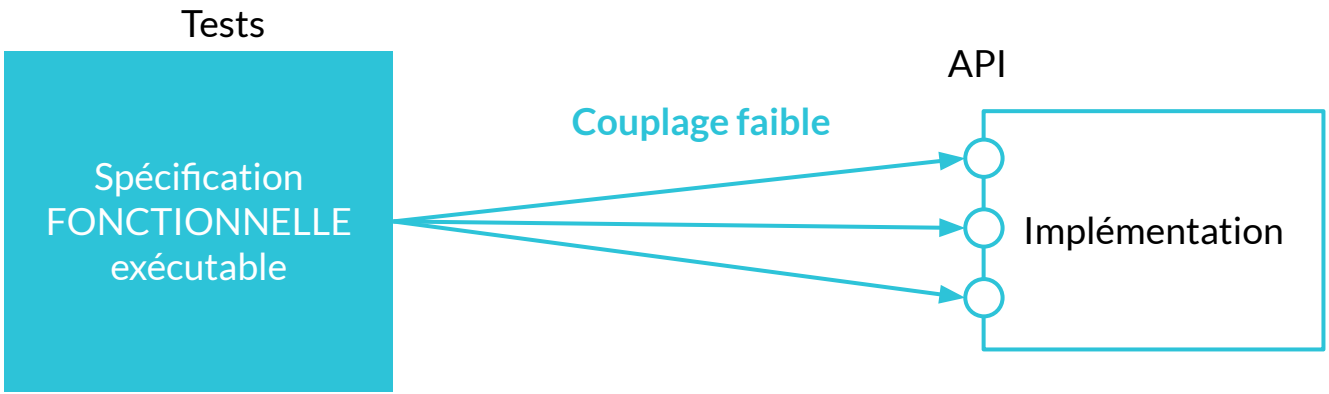
Quand on **change** les spécifications fonctionnelles, on doit également **changer** l'implémentation technique.

L'implémentation ne doit pas affecter la spécification fonctionnelle



Lorsque nous **refactorisons ou re-designons** l'implémentation de la solution, cela ne devrait **pas modifier** la spécification fonctionnelle

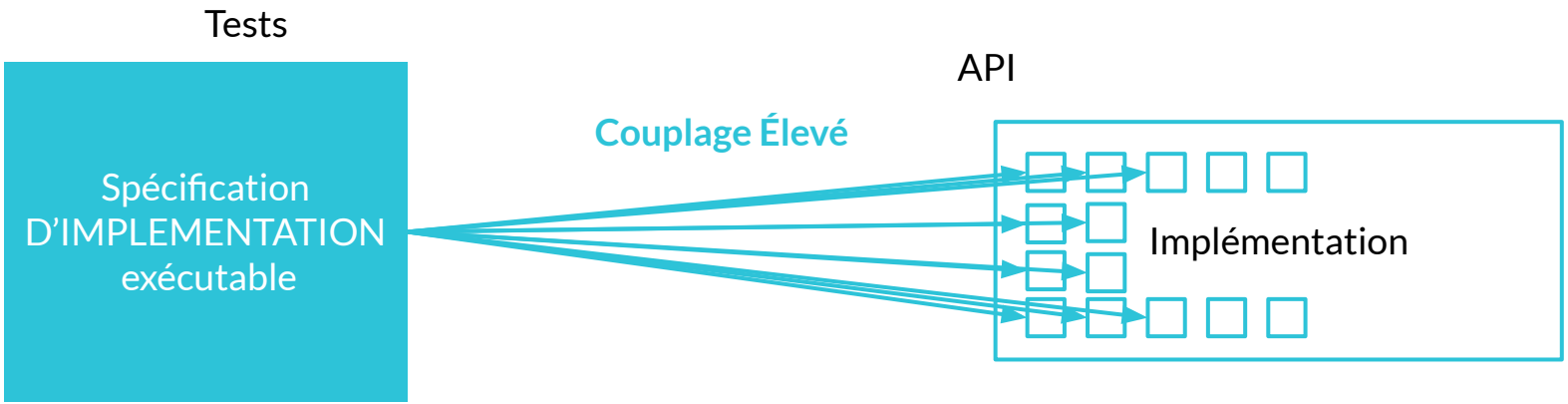
Tests en tant que spécifications requises



Les tests sont couplés à **l'API**, le **comportement externe**.

Tests robustes - modifier en toute sécurité l'implémentation interne sans modifier les tests. Les tests ne sont modifiés que lorsque les spécifications changent.

Tests en tant que spécification d'implémentation



Les tests sont couplés à **l'implémentation**, à la **structure interne**.

Tests fragiles - la modification de l'implémentation casse les tests existants, entraînant des modifications des tests même si les spécifications fonctionnelles n'ont pas été modifiées!

Résumé -Tester les spécifications fonctionnelles ou d'implémentation?

	Test = spécifications fonctionnelles	**Test = spécifications d'implémentation**
Couplage avec les tests	Couplage à l'API	Couplage à l'implémentation
Robustesse des tests	Tests robustes	Tests fragiles
Stabilité de refactorisation	Les tests sont stables	Les tests cassent
Coût de refactorisation	Aucun changement des tests	Les tests doivent être modifiés
ROI	Élevé	Faible

Qu'est-ce qu'un test unitaire?

Testons-nous le comportement des modules ou la structure des classes?

Sondage d'Audience

Quelle est votre familiarité avec la notion de tests unitaires solidaires ou solitaires ?

1. Je n'en ai pas entendu parler
2. Entendu à ce sujet, mais pas clair
3. Parfaitement familier avec elle

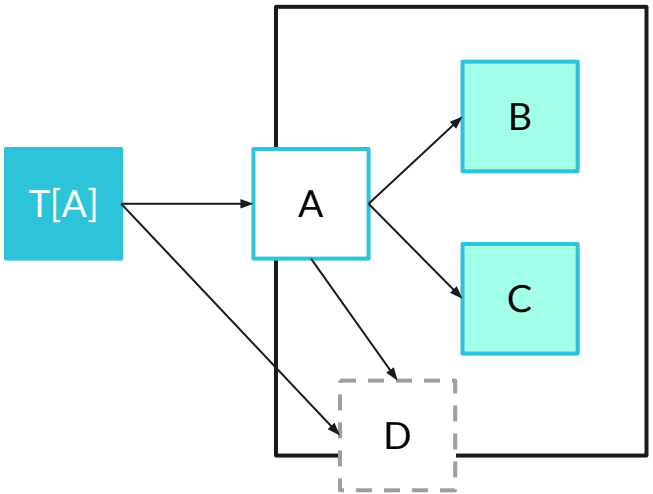
Qu'est-ce qu'un test unitaire?

- Vérifie une **unité**
- La vérifie en **isolation**
- La vérifie **rapidement**

	Tests unitaires solidaires (TDD Classique)	**Tests unitaires solitaires (TDD Mockist)**
Unité	Un module *(une ou plusieurs classes)* *(granularité élevée)*	Une classe *(granularité fine)*
Isolement	Isoler le module UNIQUEMENT des dépendances partagées (base de données, Fichiers, etc.)	Isoler la classe de TOUS ses collaborateurs

https://martinfowler.com/bliki/UnitTest.html https://freecontent.manning.com/what-is-a-unit-test-part-2-classical-vs-london-schools/

Tests Unitaires solidaires - Tester l'API du module



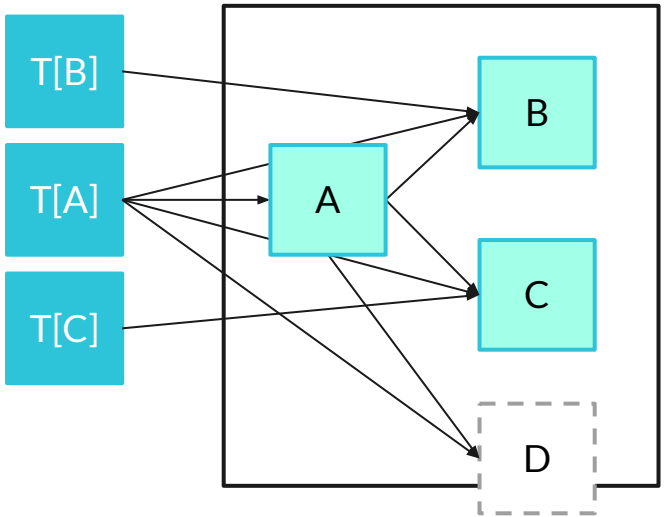
Les tests unitaires **solidaires** accèdent à **l'API du module**.

Ils ne connaissent pas les détails d'implémentation du module.

Nous utilisons des doublures de tests (test double) uniquement pour les dépendances partagées (base de données, Fichiers, etc.)

→ **Refactoriser** l'implémentation d'un module **n'a aucun impact sur les tests**.

Tests Unitaires Solitaires - Tester l'implémentation du module



Les tests unitaires **solitaires** accèdent à **l'implémentation du module**.

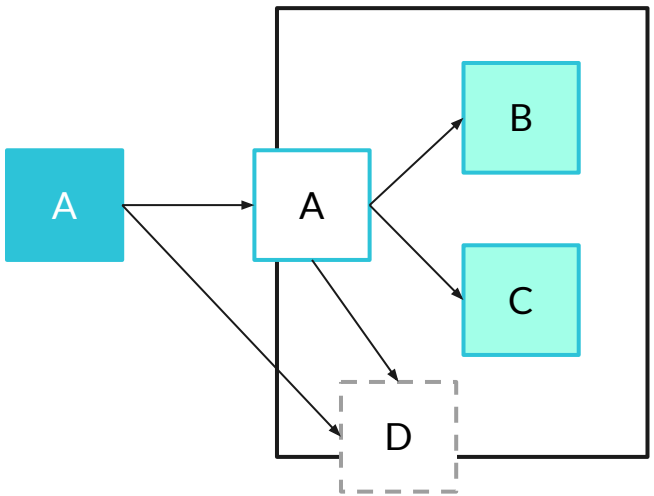
Ils connaissent les classes internes du module et leurs collaborateurs.

Nous remplaçons tous les collaborateurs par des Mocks.

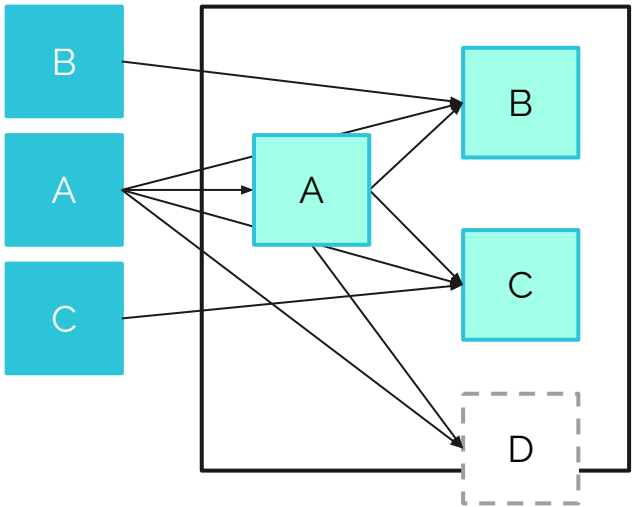
→ **La refactorisation** de l'implémentation du module **casse les tests existants**.

Micro Comparaison

Tests Unitaires solidaires

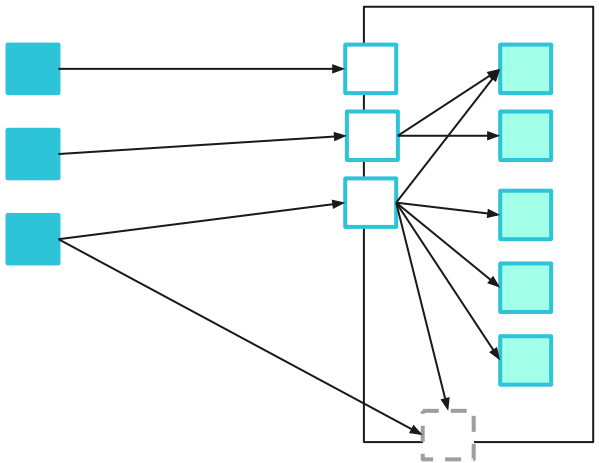


Tests Unitaires Solitaires

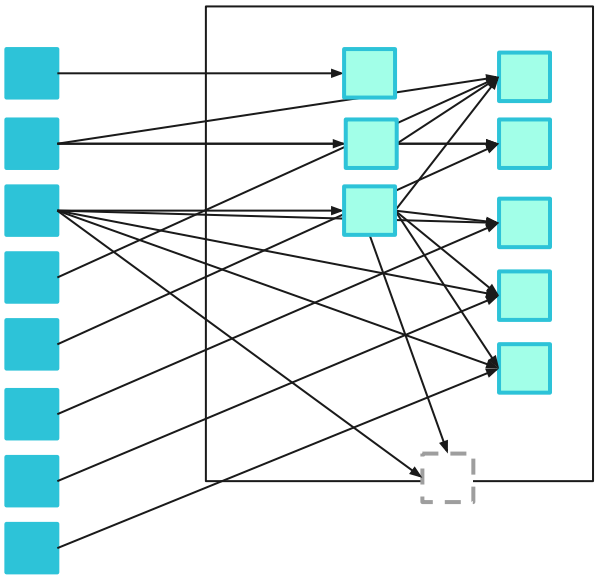


Macro Comparaison

Tests Unitaires solidaires

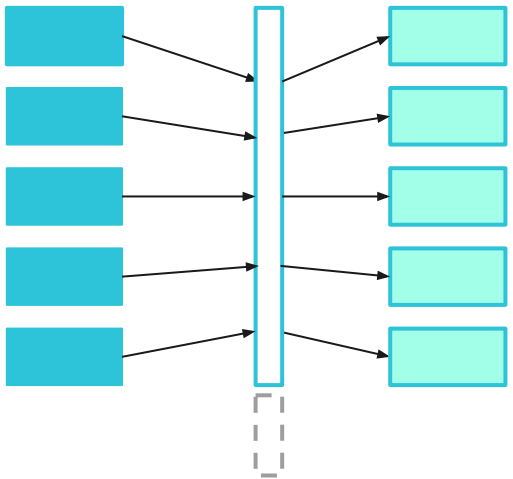


Tests Unitaires Solitaires

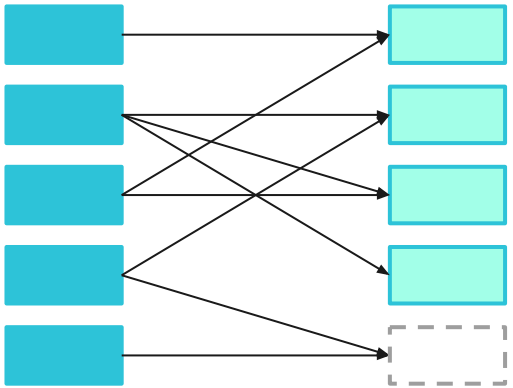


Macro Comparaison II

Tests Unitaires Solidaires
Tests couplés à l'API



Tests Unitaires Solitaires
Tests couplés à l'implémentation



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

Comparaison des Tests Unitaires

Tests unitaires solidaires (à gros-grains)	**Tests unitaires solitaires (à grain fin)**
Les tests sont couplés à l'API du module (comportement du module)	Les tests sont couplés à l'implémentation du module (structure du module)
Tests robustes → Refactorer l'implémentation du module n'impacte pas les tests	Tests fragiles → Refactorer l'implémentation du module provoque la rupture des tests
Plus peu coûteux → Moins de code de test, stabilité des tests plus élevée, coût de maintenance inférieur	Coût plus élevé → Plus de code de test, stabilité de test inférieure, coût de maintenance plus élevé

Tester Le Comportement

Les tests doivent être couplés au comportement et non à la structure

Sondage d'Audience

Quelles sont les origines du TDD et du BDD?

1. TDD était à l'origine sur les tests, et BDD était à l'origine sur le comportement
2. Le TDD et le BDD concernaient à l'origine le comportement
3. Pas vraiment sûr

Kent Beck - Tests should be coupled to behaviour

Les tests de développeur doivent être **sensibles aux changements de comportement** et **insensibles aux changements de structure**. - Kent Beck

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

Si le comportement du **programme est stable** du point de vue d'un observateur, aucun test ne devrait changer. - Kent Beck

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

Les tests doivent être **couplés au comportement** du code et **découplés de la structure** du code. - Kent Beck

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

Dan North - Behaviour Driven Development (BDD)

"Comportement" est un mot plus utile que "test" - Dan North

Les spécifictions sont à propos du comportement - Dan North

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

Dan North a tenté de "corriger" la confusion de dénomination en remplaçant le mot "test" par "comportement".
Même si beaucoup de gens associent BDD à ATDD/Gherkin/Cucumber, les origines du BDD étaient en fait une tentative de mettre en valeur l'intention comportementale du TDD.

Martin Fowler - Refactoring

Le refactoring est une technique disciplinée pour **restructurer** un code existant, en modifiant sa structure interne **sans changer son comportement externe** - Martin Fowler

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

Lorsque nous refactorons, nous changeons de structure, mais pas de comportement!

Testing at Google - “Striving for Unchanging Tests”

*“... le test idéal n’a pas à **changer**...”*

“Lorsqu'un ingénieur **refactorise les composants internes** d'un système sans modifier son interface… **les tests du système ne devraient pas avoir besoin de changer**. Le rôle des tests dans ce cas est de s'assurer que le refactoring n'a pas modifié le comportement du système.”

“**La modification du comportement existant d'un système** est le seul cas où nous nous attendons à devoir **modifier les tests**.”

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

Testing at Google - “Test via Public APIs”

“... examinons certaines pratiques pour nous assurer que les tests n'ont pas besoin de changer à moins que les exigences du système testé ne changent.”

“De loin, le moyen le plus important de s'en assurer est **d'écrire des tests** qui **invoqueraient le système** testé **de la même manière que ses utilisateurs**; c'est-à-dire faire des appels contre son **API publique** plutôt que des détails d'implémentation.”

“Si **le test repose sur le même comportement que les actions des utilisateurs**, un changement qui casse un test casse également le comportement attendu par l'utilisateur”

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

Testing at Google - “Test Behaviors, Not Methods”

“**Le premier réflexe** de nombreux ingénieurs est d'essayer de faire **correspondre la structure** de leurs tests à la structure de leur code de manière que chaque méthode de production ait une méthode de test correspondante.”

“Ce modèle peut être pratique au début, mais **avec le temps, il entraîne des problèmes.**”

“Il existe un meilleur moyen: plutôt que d'écrire un test pour chaque méthode, **écrivez un test pour chaque comportement.**”

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

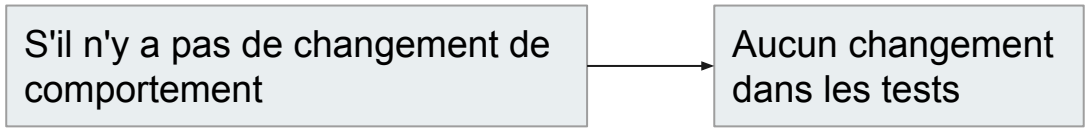
Écrire un nouveau test ou modifier un test existant?

CHANGEMENTS DE COMPORTEMENT

Exigences métier nouvelles ou modifiées

CHANGEMENTS STRUCTURELS

Refactorisation ou refonte



TDD vs TLD

Comment pilotons-nous le développement grâce à des spécifications exécutables?

Sondage d'Audience

Avez-vous essayé TDD? Quelle a été votre expérience avec TDD?

1. Jamais essayé TDD
2. Essayé TDD, mais pas convaincu
3. Essayé et partiellement adopté TDD
4. TDD essayé et entièrement adopté

TDD Rouge-Vert-Refactor

Écrire un test qui ne passe pas, pour garantir que c'est le code qu'on va écrire qui le fait passer

Écrire juste assez de code pour faire passer le test, on obtient du code qui “marche”

Nettoyer le code grâce au refactoring, on obtient un “code propre”

Boucles de Rétroaction TDD

1. TESTABILITÉ DES SPÉCIFICATIONS : Pouvons-nous rédiger un test pour la spécification?

2. FALSIFIABILITÉ DU TEST : Voyons-nous le test échouer? L'étape **ROUGE**.

3. CONCEPTION DE L'INTERFACE : L'interface est-elle pratique à utiliser? Le test est le premier consommateur.

4. EXACTITUDE DE L'IMPLÉMENTATION : Le code fonctionne-t-il? L'étape **VERTE**.

5. QUALITÉ DE LA MISE EN ŒUVRE : La mise en œuvre est-elle propre? L'étape de **REFACTORISATION**.

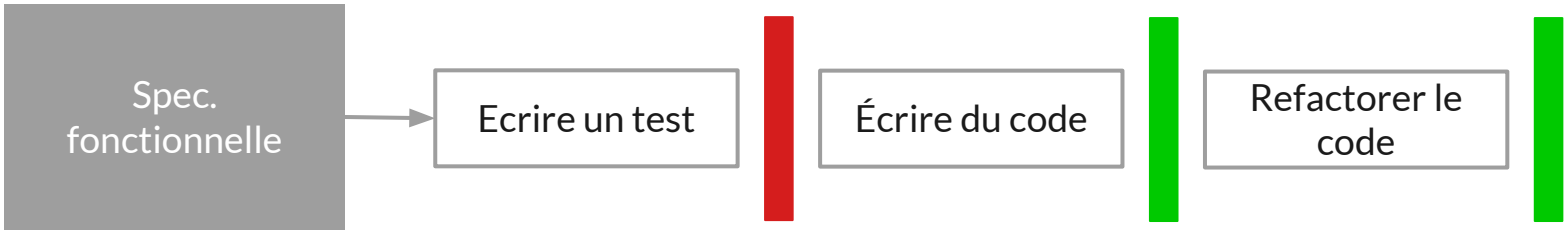
Sondage d'Audience

À quel moment votre équipe rédige-t-elle des tests unitaires?

1. Nous n'écrivons pas du tout de tests unitaires parce que mon équipe ne veut pas
2. Nous n'écrivons pas de tests unitaires car nous n'avons pas le budget / le temps pour cela
3. Nous écrivons d'abord du code, puis écrivons les tests unitaires par la suite
4. Nous écrivons toujours le test unitaire en premier, puis écrivons du code après le test

Test Driven Development

TDD se traduit par un développement **plus rapide** grâce à une boucle de rétroaction plus **courte**



1. Spec testable?
2. Le test peut-il échouer?
3. Interface facile à utiliser?

4 Est-ce que l'implémentation fonctionne?

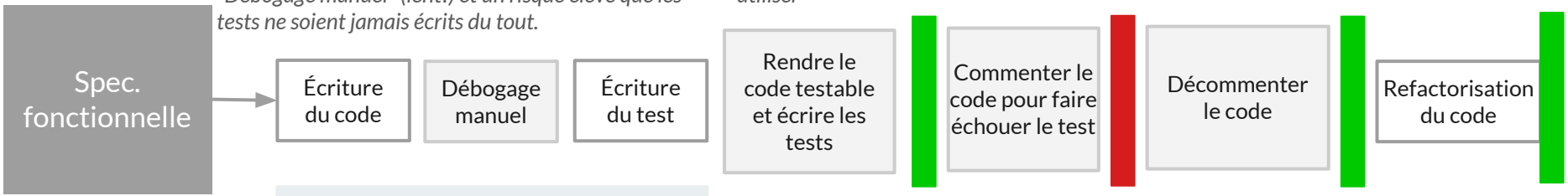
5. Est-ce que le code de l'implémentation est propre?

Test Last Development

TLD entraîne un développement plus **lent** en raison d'une boucle de rétroaction plus **longue**

*Dans le **meilleur des cas TLD**, il n'y a pas de "débogage manuel". Dans le **pire des cas TLD** (commun!), il y a "Débogage manuel" (lent!) et un risque élevé que les tests ne soient jamais écrits du tout.*

Assure que le code est testable et que l'interface est simple à utiliser



1. Spec. fonctionnelle testable?
2. Test falsifiable?
3. Interface simple à utiliser?
4. Est-ce que l'implémentation fonctionne?
5. Est-ce que le code est propre?

TDD vs TLD - Résumé

TDD se traduit par un développement plus **rapide** grâce à une boucle de rétroaction plus **courte**

TDD **garantit** que le code est couvert par des tests (car nous n'écrivons jamais de code sans tests au préalable)

TLD entraîne un développement plus **lent** en raison d'une boucle de rétroaction plus **longue**

TLD **ne garantit pas** que le code sera couvert par des tests (dans le pire des cas, les tests peuvent ne jamais être écrits)

TDD et Clean Architecture

Piloter l'architecture des applications à travers le comportement du système

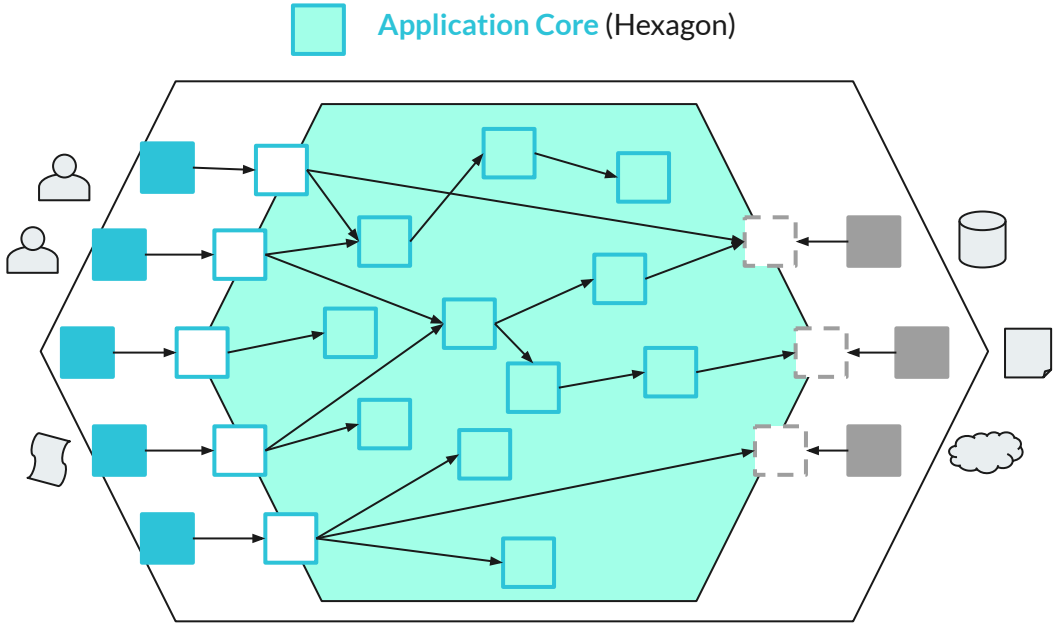
Sondage d'Audience

Votre équipe utilise-t-elle l'une de ces architectures?

1. CRUD - Contrôleurs, Services, Entités (ORM), Repositories (ORM)
2. Architecture Hexagonale
3. Onion Architecture
4. Clean Architecture
5. Autre chose

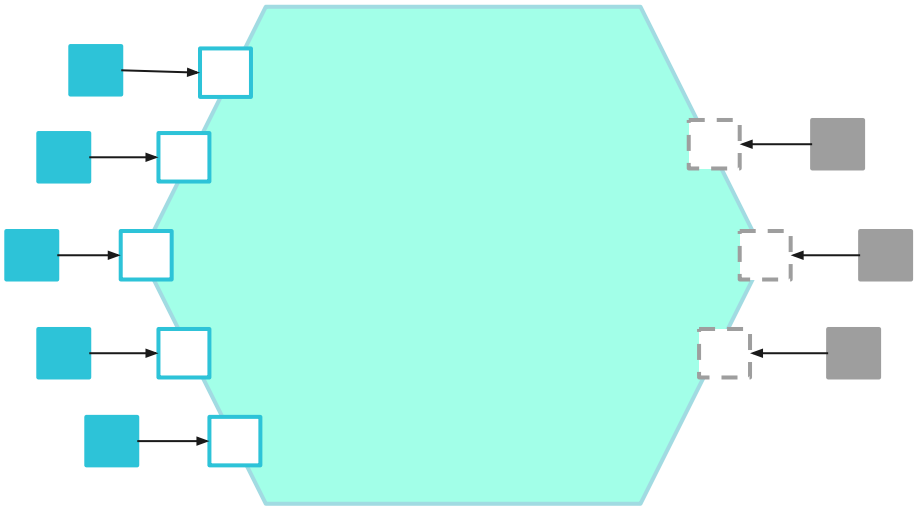
https://alistair.cockburn.us/hexagonal-architecture/

Hexagonal Architecture



Hexagonal Architecture - Tests Unitaires

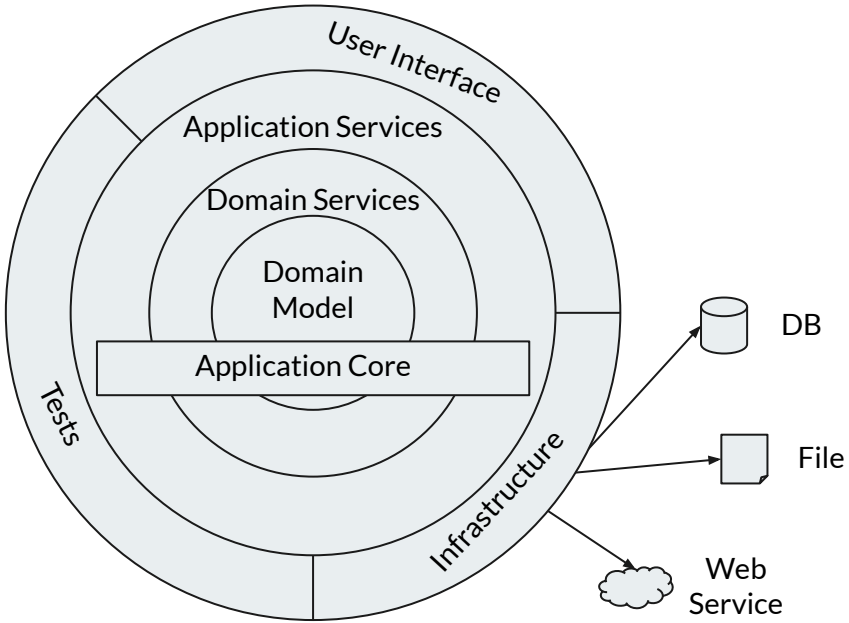
Les tests unitaires peuvent exécuter des cas d'utilisation du système via la **user-side API**



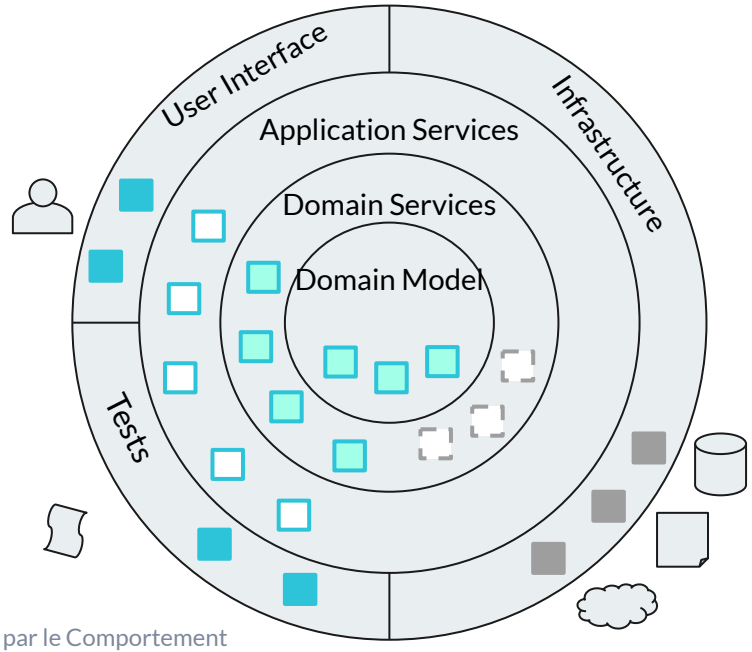
Les doublures de tests servent d'adaptateurs en mémoire pour la **server-side API**

Onion Architecture

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



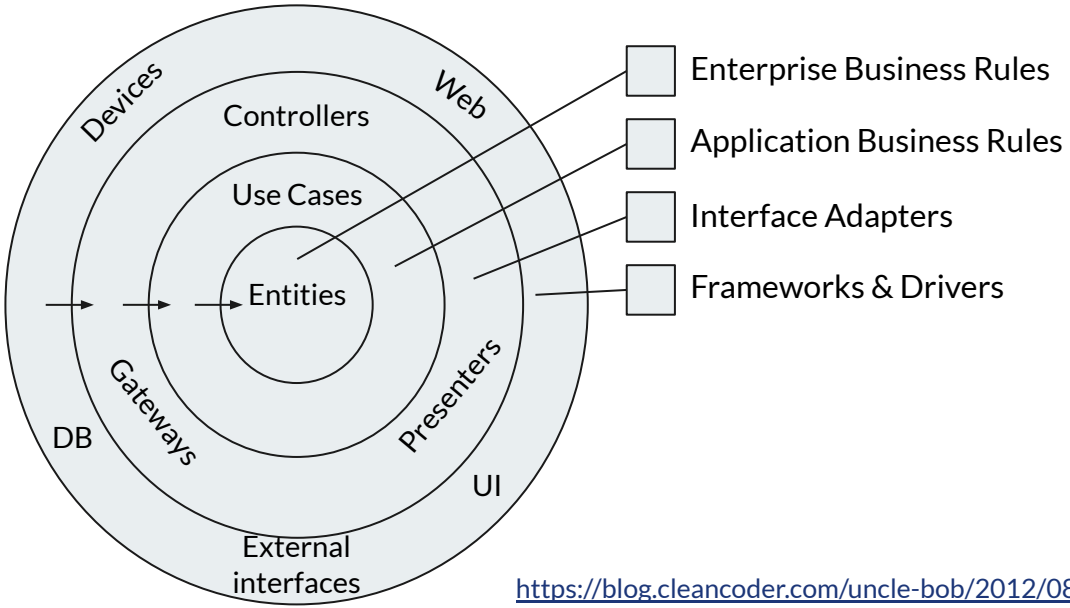
Onion Architecture & Hexagonal Architecture

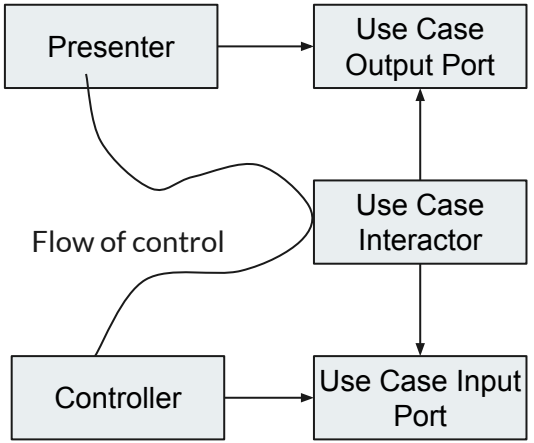


Hexagonal Architecture

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

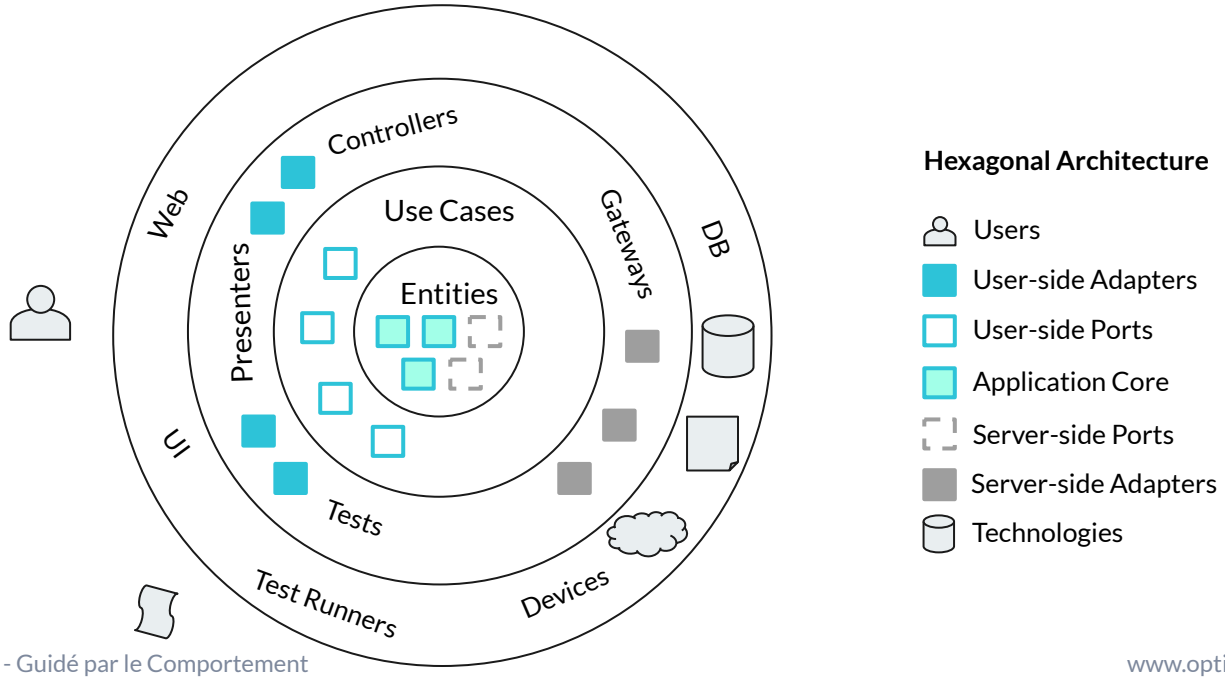
Clean Architecture





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

Clean Architecture & Hexagonal Architecture



Architectural Equivalence

System under Test (SUT)

- Users
- SUT Tests
- SUT API
- SUT Implementation
- Shared Dependency Interfaces
- Shared Dependency Implementations
- Technologies

Hexagonal Architecture

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

Onion Architecture

- Users
- UI, Tests
- Application Services
- Domain Model
- Domain Services
- Infrastructure
- Technologies

Clean Architecture

- Users
- Presenters, Tests
- Use Cases
- Entities
- Gateway Interfaces
- Gateways
- Technologies

Tests d'acceptation - Tests agissant en tant qu'utilisateurs

Tests d'acceptation - Niveau Unitaire

Les tests unitaires exécutent des cas d'utilisation (use case)

Les dépendances partagées sont remplacées par des doublures de tests

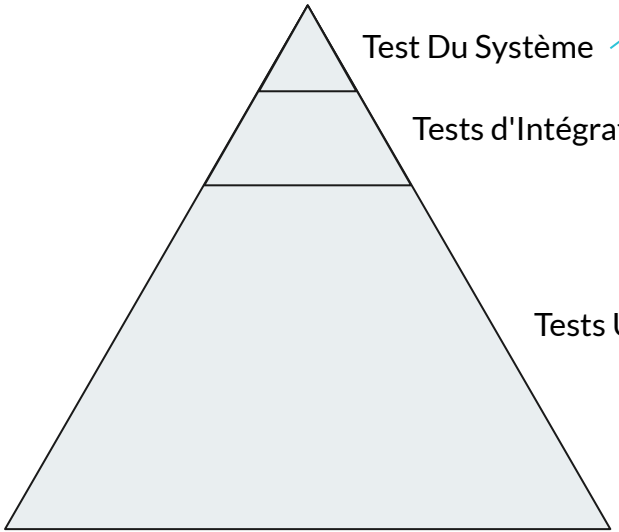
Tests d'acceptation - Niveau E2E

Les runners de l'interface utilisateur exécutent des cas d'utilisation (use case)

Les dépendances partagées sont remplacées par des implémentations réelles

Avantage: nous pouvons exécuter des tests d'acceptation au niveau de l'unité via les cas d'utilisation, comme l'utilisateur! Rétroaction et couverture des scénarios beaucoup plus rapides au niveau unitaire

Test Pyramid Summary



Test Du Système

Tests d'Intégration

Tests Unitaires

- SUT Tests
- SUT API
- SUT Implementation
- Shared Dep. Interfaces
- Shared Dep. Implementations
- Technologies

- Shared Dep. Tests
- Shared Dep. Interfaces
- Shared Dep. Implementations
- Technologies

- SUT Tests
- SUT API
- SUT Implementation
- Shared Dep. Interfaces
- Shared Dep. Test Doubles

Conclusion

Les tests doivent être des spécifications d'exigence exécutables… pas des spécifications d'implémentation

Les tests doivent être couplés à l'API… pas à l'implémentation

Les tests doivent être couplés au comportement… pas à la structure

La Clean Architecture expose des use cases, nous pouvons tester le comportement de l'application

Le refactoring ne modifie pas le comportement, n'affecte pas les tests comportementaux

Les tests comportementaux sont plus robustes et ont un coût de maintenance plus faible.

Merci

Valentina Cupać @ Optivem

Connectez-vous ou **suivez-moi** sur LinkedIn pour en savoir plus sur TDD et Clean Architecture

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

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

Remerciements

Merci aux personnes suivantes pour leur précieux soutien dans la traduction de cette présentation en français:

- Edouard Mangel
- Pierre Criulanscy