

Pratiques

Approval testing

- [C'est quoi ?](#)
- [Objectif](#)
- [Mise en place sur un projet](#)
 - [En .Net avec Verify et Xunit](#)
 - [En Javascript avec Jest](#)
- [Bonnes pratiques](#)
 - [Les fichiers de référence doivent être intégrés dans le versioning](#)
 - [Le contenu des éléments testés par approval doit être déterministe](#)

C'est quoi ?

"**Approval tests** simplify unit testing this by taking a snapshot of the results, and confirming that they have not changed."

Voir [Approvaltests.com](#).

L'approval testing est aussi connu sous le nom de golden master testing, snapshot testing ou plus rarement de characterization testing.

Objectif

- Simplifier les asserts d'objets / de structures complexes
- Mettre rapidement en place un premier niveau de couverture de test sur du code non testé

Mise en place sur un projet

En .Net avec Verify et Xunit

Documentation de Verify: <https://github.com/VerifyTests/Verify>

Installer le nugget Verify.Xunit: <https://www.nuget.org/packages/Verify.Xunit/>

dotnet add package Verify.Xunit

Dans le projet de test, modifier le **.csproj** et autoriser l'`ImplicitUsings` dans le `PropertyGroup` (nécessaire au bon fonctionnement de Verify avec la syntaxe présentée) :

```
<PropertyGroup>
  <ImplicitUsings>enable</ImplicitUsings>
</PropertyGroup>
```

Si on prend un test qui assert un objet en output d'une manipulation (exemple trivial ici):

```
public class InventoryTest
{
    [Fact]
    public void item_should_have_valid_values_at_init()
    {
        var item = new Item { Name = "foo", SellIn = 0, Quality = 0 };
        Assert.Equal("foo", item.Name);
        Assert.Equal(0, item.SellIn);
        Assert.Equal(0, item.Quality);
```

```
}
```

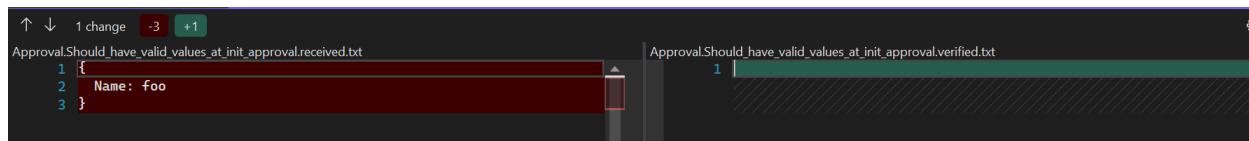
On peut le simplifier en mettant en place une validation via de l'approval testing:

- Ajouter l'annotation [UsesVerify] à la classe de test
- Changer le type de retour du test, de void à Task
- Supprimer les assert et retourner le résultat du Verify de notre item

```
[UsesVerify]
public class InventoryTest {
    [Fact]
    public Task item_should_have_valid_values_at_init()
    {
        var item = new Item { Name = "foo", SellIn = 0, Quality = 0 };
        return Verify(item);
    }
}
```

Le premier run du test tombera en échec. C'est normal, on a pas encore de fichier de référence.

Un diff s'ouvre, normalement automatiquement, pour comparer les deux fichiers: le **.received**, généré lors de ce test, et le **.verified**, qui est notre source :



Si le résultat nous semble correct, on peut directement copier/coller le contenu du **.received** dans le **.verified**.

Après cela, chaque fois que le test sera exécuté, le résultat sera comparé à ce fichier **.verified**. Le diff s'ouvrira automatiquement lorsqu'une différence sera relevée entre les deux fichiers.

Sur cet exemple précis, on remarque que les valeurs de `SellIn` et `Quality` ne ressortent pas dans le fichier.

C'est le comportement du `Serializer` utilisé par `Verify`, qui ignore les valeurs par défaut: 0 pour un entier, null pour un objet, ...

On peut forcer le rendu de ces champs en configurant le `Serializer` ([Documentation ici](#))

```
[UsesVerify]
public class InventoryTest
{
    public InventoryTest()
    {
        VerifierSettings.ModifySerialization(settings =>
            settings.AddExtraSettings(serializerSettings =>
                serializerSettings.DefaultValueHandling = DefaultValueHandling.Include));
    }

    [Fact]
    public Task item_should_have_valid_values_at_init()
    {
        var item = new Item { Name = "foo", SellIn = 0, Quality = 0 };
        return Verify(item);
    }
}
```

On voit alors bien apparaître tous nos attributs:

```

↑ ↓ 1 change -5 +1
Approval.Should_have_valid_values_at_init_approval.received.txt
1 {
2   Name: "foo",
3   SellIn: 0,
4   Quality: 0
5 }

```

Approval.Should_have_valid_values_at_init_approval.verified.txt

En Javascript avec Jest

Documentation des snapshots dans Jest: <https://jestjs.io/docs/snapshot-testing>

Ici nous n'avons besoin que de Jest. Donc si ce n'est pas déjà fait, la seule commande à lancer est :

```
npm install --save-dev jest
```

Si on prend un test qui assert un objet en output d'une manipulation (exemple trivial ici):

```

const Item = require("../src/Item")

describe("Freshly created item", () => {
  it("should have valid values", () => {
    let item = new Item("foo",0,0);

    expect(item.getName()).toBe("foo")
    expect(item.getSellIn()).toBe(0)
    expect(item.getQuality()).toBe(0)
  });
})

```

On le transforme facilement en approval test avec toMatchSnapshot:

```

const Item = require("../src/Item")

describe("Freshly created item", () => {
  it("should have valid values", () => {
    let item = new Item("foo",0,0);

    expect(item).toMatchSnapshot();
  });
})

```

La première fois que le test se lance, il n'y a pas de fichier de référence (snapshot dans Jest). On obtient alors le message :

```

Freshly created item
  ? should have valid values (2 ms)

> 1 snapshot written.
Snapshot Summary
> 1 snapshot written from 1 test suite.

```

Un dossier `__snapshots__` est automatiquement créé, et on voit apparaître un fichier avec le contenu suivant :

```

exports[`Freshly created item should have valid values 1`] =
Item {
  "name": "foo",
  "quality": 0,
  "sellIn": 0,
}
``;

```

À partir de ce moment, chaque fois que le test sera rejoué, la sortie du test sera comparée à ce fichier.

En cas de différence, on obtient un diff dans le terminal :

```
Freshly created item
  ✘ should have valid values (3 ms)

● Freshly created item > should have valid values
  expect(received).toMatchSnapshot()

  Snapshot name: `Freshly created item should have valid values 1`  

  - Snapshot - 1
  + Received + 1

  Item {
    "name": "foo",
    "quality": 0,
    - "sellIn": 0,
    + "sellIn": 1,
  }
```

Si besoin, le fichier snapshot peut facilement être mis à jour via le menu interactif, qui apparaît lorsque l'on lance les tests en **--watch** :

```
> Press u to update failing snapshots.
> Press i to update failing snapshots interactively.
```

u va automatiquement mettre à jour le fichier de référence avec la sortie actuelle.

i propose un prompt pour éditer le snapshot, mais le prompt ne pop pas forcément selon le type de terminal que vous utilisez.
Pour des modifications ponctuelles, vous pouvez aussi éditer directement le fichier du dossier `__snapshots__`

Bonnes pratiques

Les fichiers de référence doivent être intégrés dans le versioning

Cela concerne les `.approved` pour Verify ou le contenu du dossier `__snapshots__` pour Jest. Ces fichiers doivent être traités comme un élément à part entière de votre base de code.

Pour Verify, il est conseillé d'ajouter la ligne suivante au `.gitignore`, les fichiers `.received` n'ayant aucun intérêt:

```
*.received*
```

Le contenu des éléments testés par approval doit être déterministe

Comme on fait une comparaison textuelle, il est nécessaire que les mêmes actions entraînent le même résultat, et que l'on valide donc le fichier de référence.

Il est tout de même possible de faire de l'approval testing sur des sorties ayant une partie d'aléatoire, en ignorant les parties aléatoires.

Dans Verify, cela se fera avec les [Scrubbers](#). Avec Jest, pas de mécanique dédiée, on pourra supprimer les éléments problématiques avant le expect.

Architecture Hexagonale

- [Qu'est-ce que l'Architecture hexagonale ?](#)
- [Les grands principes de cette architecture](#)
- [Quand est-ce qu'elle est pertinente à utiliser ?](#)
- [Gestion de la complexité dans le code](#)
- [Les avantages à l'utiliser](#)
- [Un peu de code pour visualiser les propos](#)
- [Les inconvénients de cette architecture](#)
- [Conclusion](#)
- [Ressources](#)

Qu'est-ce que l'Architecture hexagonale ?

Alistair Cockburn a publié en 2005 les principes de cette architecture, qui a pour principal but d'éviter les dépendances indésirables entre les couches d'une application. Par exemple les considérations de l'interface utilisateur avec la logique et règles métier ou encore avec le système de persistence. En gros c'est une architecture qui permet de protéger son domaine métier de la technique, quelle qu'elle soit.

Cet objectif est partagé par d'autres architectures, qui vont avoir des caractéristiques et des terminologies différentes, mais qui au fond mènent le même combat ; celui de protéger le métier de la technique le plus possible.

On pourra donc entendre parler de Clean architecture, Onion architecture ou encore architecture Ports and Adapter.

Quand est-ce qu'elle est pertinente à utiliser ?

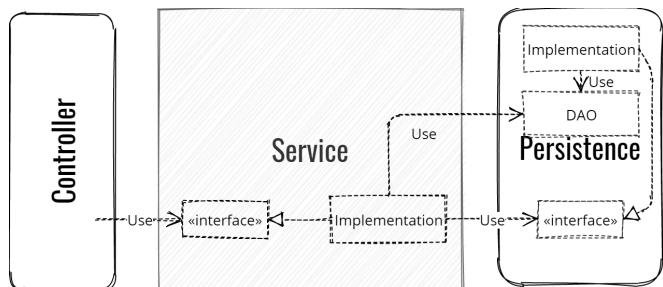
Si on se pose la question de la pertinence de l'utiliser, on comprend tout de suite qu'il y a effectivement des endroits où cette architecture n'est pas à son avantage. L'application systématique de cette architecture sur tout le périmètre d'une application (voir SI) lui a parfois valu une mauvaise presse, de par sa complexité et verbosité. Mais il en va de l'équipe mettant en place cette architecture, de se poser la question de la pertinence de cette architecture sur son périmètre fonctionnel.

Car effectivement, c'est la complexité du périmètre fonctionnel qui va en partie être responsable du choix de cette architecture, complété bien sûr avec la complexité de l'environnement technique.

L'une des variables à prendre en compte absolument est donc de savoir si on a réellement un métier complexe à encapsuler dans notre code, qui doit être pérennisé sur le long terme et donc doit être facilement maintenable et modifiable.

Les grands principes de cette architecture

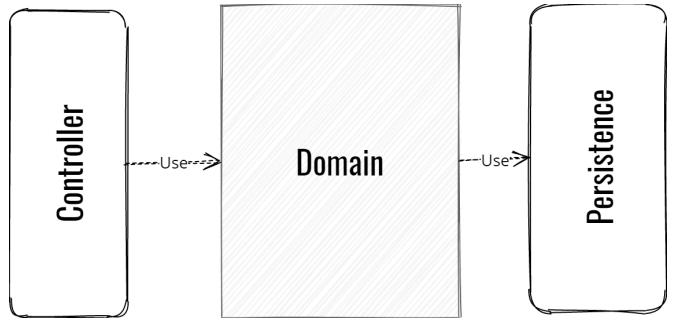
Voyons rapidement ensemble les grandes lignes de cette architecture. Afin de mieux comprendre l'apport des caractéristiques proposées par l'architecture hexagonale, revenu sur une application en couche, comme on peut le voir sur la majorité des projets.



Dans ce style d'architecture classiquement on retrouve un contrôleur qui passe la main à un service qui lui-même interroge une partie persistance pour retrouver / stocker de la donnée. Dans ce cas de figure, la plupart du temps le modèle objet est anémique, avec uniquement des getters, setter et le logique métier se trouve à diluer dans tous les couches de l'architecture.

Dans l'approche de l'architecture hexagonale, le concept de ségrégation des responsabilités est central, notamment vis-à-vis de l'isolation de du logique métier ; qui est centralisé dans ce qu'on appelle un *domaine*.

Il faut prendre en compte (ce qui est plus rarement le cas en vérité), le changement de technologie liée à notre système de persistance ou d'appel de notre service. Mais aussi des Frameworks internes à l'application nous aidant à délivrer le besoin de manière plus rapide.



Gestion de la complexité dans le code

Avant de rentrer dans les détails de l'implémentation technique, petit rappel sur la complexité d'un logiciel, qui peut donner aussi des indications sur quand est-ce qu'il pertinent d'utiliser cette architecture.

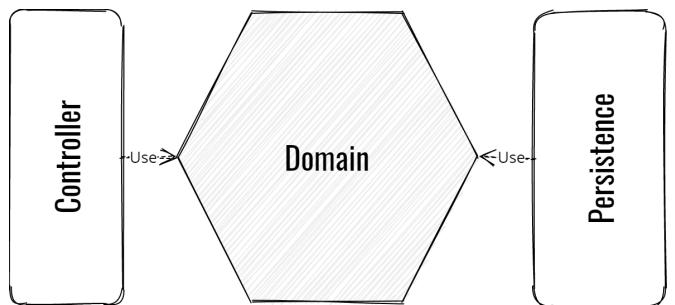
Le code d'un logiciel encapsule 3 types de complexité différents, la complexité: Essentielle, Obligatoire et Accidentelle.

- **L'essentielle**, est la complexité liée à votre métier. C'est la complexité intrinsèque de votre métier, la complexité et richesse des règles fonctionnelles. C'est tout le code lié à la modélisation de tout ça, une fois le scope de la fonctionnalité décidé par le métier, il y aura forcément du code pour la modéliser.
- **L'obligatoire**, c'est la complexité liée à la technique minimum pour servir la fonctionnalité à nos utilisateurs, via web browser, apps mobiles ou autres on aura besoin de gérer un serveur web. Si on veut que nos utilisateurs retrouvent leurs données entre deux connexions, il faudra bien un système de persistance, donc potentiellement on devra gérer une base de données. Cela conduit forcément du code pour communiquer avec cette technologie.
- **L'accidentelle**, qui théoriquement ne devrait pas exister, mais qui est bien présente dans toutes les codes base, on ne peut pas la faire disparaître, mais on peut tenter de la diminuer si on est conscient qu'elle existe. Dedans nous allons retrouver le mauvais usage d'un langage, d'un Framework, de l'application de pattern non approprié, la sur-ingénierie etc... et aussi le fait que la complexité Essentielle et Obligatoire soient fusionnés ensemble.

Tout l'objectif de l'architecture hexagonale est de séparer ses deux complexités, afin de pouvoir les appréhender distinctement, cela à pour effet de faire baisser la complexité Accidentelle par design.

Jusque-là pas de grande différence notable visuellement sur l'architecture, même si du domaine aura tendance à être le focus use case que service, comme on verra par la suite.

Une autre notion centrale, c'est l'inversion des dépendances, car le domaine ne doit dépendre que de lui-même pour rester à l'égard de la complexité provenant de la technique. Technique qui sera gardée à l'extérieur qu'on appelle *Infrastructure*.



Les avantages à l'utiliser

Maintenabilité

Le code du domaine pouvant être modifié et étendu en toute sécurité, grâce au filet de sécurité

Pour que le domaine ne dépende d'aucun framework, il va falloir déclarer des contrats aux limites du domaine pour décrire ce qu'on peut apporter comme service d'un côté et ce qu'on attend de nos collaborateurs externes de l'autre.

des tests en isolation du domaine, la maintenabilité de l'application est forcément que si tout était couplé.

■ Lisibilité

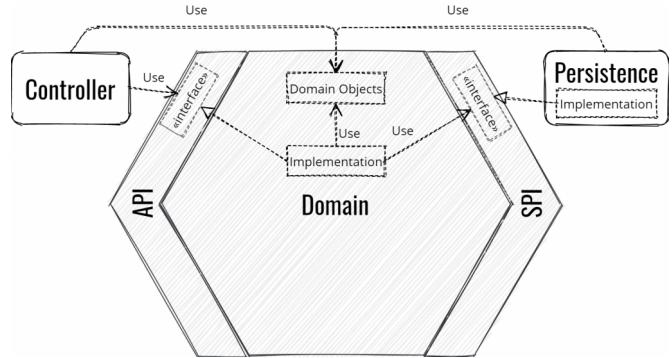
Le fait que le logique métier / règle de gestion soit séparée des considérations techniques, le code est plus lisible.

■ Testabilité

L'application étant plus modulaire, il est facile de monter l'application à la carte, comme des legos pour assembler différentes parties qu'on voudrait tester séparément. Par exemple, que les contrôler avec un service mocké, que les services métier avec les systèmes externes mocké, que l'implémentation d'un contrat de SPI (une repository implémentée avec Mongo par exemple) ou tout assemble pour faire un grand test d'intégration etc.

■ Robustesse

Le fait de tout contrôler à la carte et la testabilité rend le code robuste, cela évite les effets indésirables à l'autre bout du système lorsqu'on fait des modifications à un endroit du code.



On retrouve au centre du schéma ci-dessus le centre de cette architecture est appelé le Domaine (API et SPI font partie du Domaine aussi) et tout ce qui est à l'extérieur est plus communément appelé Infrastructure. L'API métier, représente le contrat de service fourni et implémenté par le Domaine, l'aspi est l'interface (qui peut être multiple) décrivant le comportement attendu des services externes (au Domaine) afin de pouvoir assurer son métier.

Un peu de code pour visualiser les propos

Rien de mieux que du code pour comprendre un principe !

Vous trouverez dans ce repository en cours de construction une implémentation de l'architecture hexagonale sur le kata Banking:

<https://github.com/valentinacupac/banking-kata-dotnet>

Une version meetup de la création de cette codebase est disponible ici:

<https://www.youtube.com/watch?v=IZWLnn2fNko>

N'hésitez pas à regarder comment les tests sont fait !

Les inconvénients de cette architecture

Le corollaire du point 2, c'est évidemment d'utiliser cette architecture lorsqu'il n'y a pas de métier complexe à encapsuler dans le code.

Ce n'est pas un inconvénient direct de l'architecture, mais une dérive qui arrive malheureusement fréquemment et qui est mis sous le coût de l'architecture hexagonale, alors qu'il en est rien.

Ses interfaces ont la particularité de n'utiliser uniquement que les objets déclarés dans le domaine ou le langage natif.

Il revient donc à l'infrastructure d'utiliser l'API métier via le contrôler et d'adapter les requêtes http-en-objet du domaine ou la couche de persistance d'adapter les objets stockés en base de données en objet du domaine.

Les couches d'infrastructure jouent le rôle d'anti-corruption layer, en s'assurant que les "services" du domaine peuvent prendre des décisions métier, uniquement avec des objets valide, fonctionnellement parlant. Le fait que les services métier dépendent d'interface et pas d'implémentation concrète permet de venir tester le domaine en isolation de manière facile. Il est donc possible d'avoir une couverture de 100% en test unitaire

Pour une application qui transforme une donnée en un autre format, par exemple, cela peut même être contre-productif et inapproprié de l'utiliser. Il en va de même pour les applications qui sont réellement orienté CRUD, ce qui peut être visible par une utilisation de certains outils du DDD Stratégique.

Les 2 inconvénients majeurs sont:

- La transition dans le domaine d'erreur technique malgré le découplage créé dans le code. Effectivement le domaine ne dépend pas d'implémentation concrète, mais d'interface décrivant le comportement attendu d'une implémentation, mais néanmoins au runtime certaines exceptions qui n'est pas pertinent de catcher dans le domaine peut le traverser. Certaines exceptions pour lesquelles il n'est pas pertinent de prendre une décision métier peuvent traverser malgré le découplage mis en place et doivent être catché de l'autre côté du domaine (par l'utilisateur de l'API du domaine)
- De la logique métier peut pousser dans les adapters. L'objectif de cette architecture est de centraliser les règles de gestion et orchestration de brique dans le domaine pour que le métier soit maintenable dans le temps, mais il n'est pas toujours facile de tenir cette règle, notamment pour des raisons de performance. Il peut arriver qu'on mette beaucoup de métier dans les requêtes de recherche en base de données directement ou côté implémentation de SPI pour ne pas monter en mémoire trop de chose. Cela va parfois à l'encontre de l'objectif cité plus haut.

Un dernier inconvénient caché est la montée en compétence des nouveaux arrivants qui ne connaissent pas cette architecture. Effectivement l'architecture n-tiers, en mode CRUD est la plus fréquente dans industrie, il peut être parfois compliqué pour des développeurs non familier avec les objectifs de cet architecture de comprendre l'intérêt et du coup de rentrer dedans correctement. D'où l'intérêt des ADRs ou des tests d'architecture, pour expliquer les grandes lignes et le pourquoi de ce choix à la base.

Conclusion

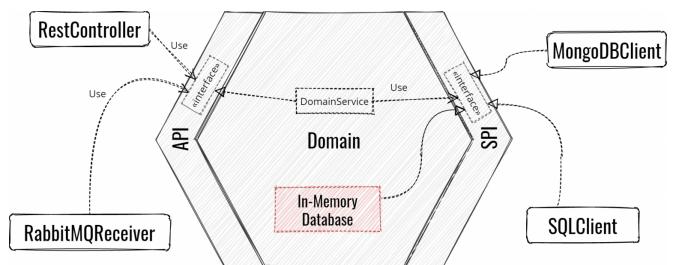
Cette architecture n'est clairement pas une "silver bullet" et doit être utilisée en pleine conscience de l'équipe qui devra maintenir le projet. Les mappings en entrée et sortie du domaine fait partie intégrante de cette architecture, mais cela peut très vite être subit s'il n'y a pas lieu de le faire (et qu'il n'y a pas vraiment de métier à protéger donc). L'utilisation des outils du DDD Stratégique sera d'une grande aide pour savoir si l'effort technique de mettre en place cette architecture est justifié.

L'objectif n'est pas de parler de l'implémentation d'une stratégie de test dans une architecture hexagonale, mais il est clairement plus facile d'avoir une batterie de tests unitaires efficiente. La modularité qu'apporte cette architecture vous permettra de réaliser techniquement, de manière assez facile, tous les niveaux de tests souhaités (Test fonctionnel (BDD), Test d'intégration etc...)

Ressources

sur le domaine sans forcer, cela en couplant le développement du domaine avec la pratique du TDD. (Sans que ça soit l'objectif principal non plus, c'est comme un bonus !)

Cela est possible grâce à la possibilité d'avoir du coup plusieurs implémentations des SPIs. Une qui va réellement être connectée à la technique souhaitée, par exemple une base de données MongoDB ou une queue Kafka etc. et une autre implémentation "In-Memory", qui va être une implémentation simpliste qui n'a de vocation à être utilisé dans les tests (dans un premier temps...)



On peut voir aussi qu'il serait facile dans cette configuration de changer de système de persistance, car les 2 clients devront uniquement remplir le contrat pour pouvoir être iso fonctionnel avec le domaine. Aucun impact donc sur la logique métier quand on change de technologie de base de données !

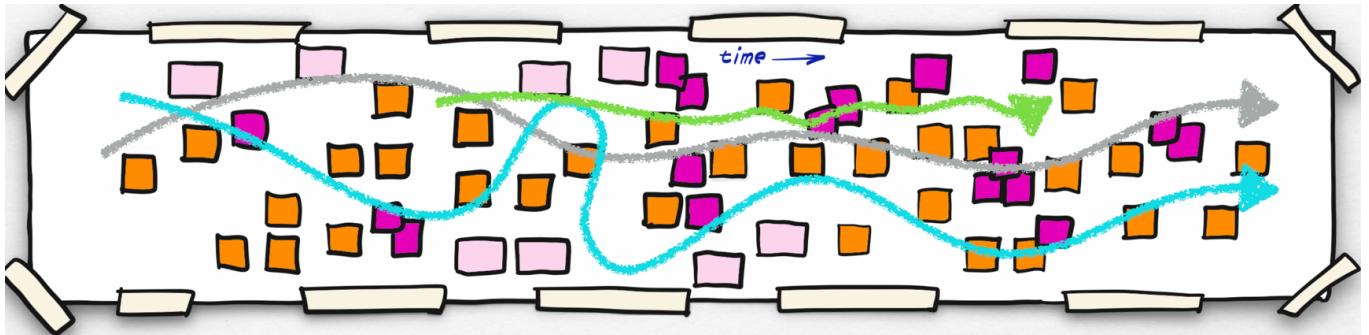
<https://www.lilobase.me/certaines-complexites-sont-plus-utiles-que-dautres/>

[https://fr.wikipedia.org/wiki/SOLID_\(informatique\)](https://fr.wikipedia.org/wiki/SOLID_(informatique))

<https://beyondxscratch.com/fr/2018/09/11/architecture-hexagonale-le-guide-pratique-pour-une-clean-architecture/>

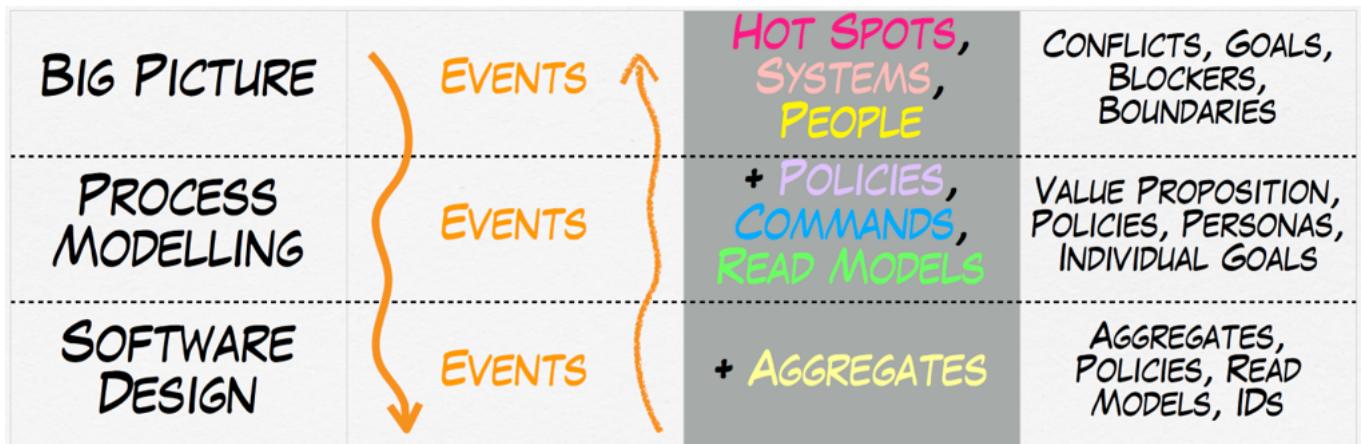
Event Storming

- [Différents niveaux d'abstraction pour différents objectifs](#)
 - [Big picture event storming](#)
 - [Process modeling event storming](#)
 - [Software design event storming](#)
- [Déroulement](#)
 - [Étape 1 : projeter les événements](#)
 - [Étape 2 : ajouter les actions et les acteurs associés](#)
- [Ressources](#)



L'Event Storming est une approche créée par Alberto Brandolini permettant de vous aider à modéliser votre domaine métier en équipe.

Différents niveaux d'abstraction pour différents objectifs



Big picture event storming

- Comprendre à gros grain l'espace du problème et aligner la vision de l'équipe
- Séparer les responsabilités au sein d'un produit / d'une organisation
- Déduire un Ubiquitous Language, langage du métier qui sera diffusé jusque dans le code

Process modeling event storming

- Identifier les flux découlant d'actions utilisateurs
- Déduire un Ubiquitous Language, langage du métier qui sera diffusé jusque dans le code

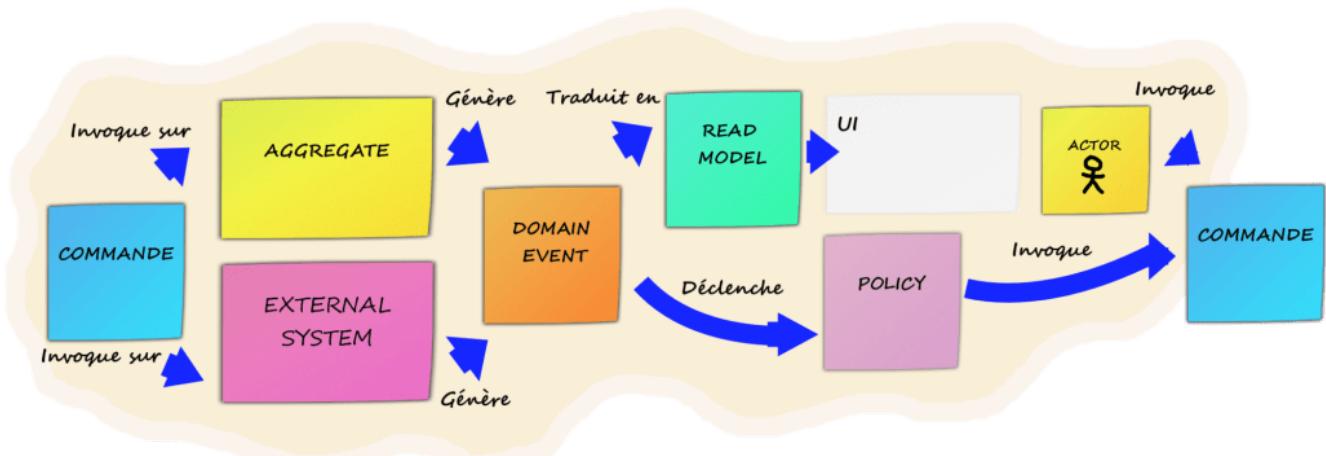
Software design event storming

- Déterminer qui (Bounded Context) représente le Core Domain
- Déterminer quels sont les agrégats de ce Core Domain

Organisation

1. Trouver la bonne salle pour réunir tout le monde (et être libres de mouvements) : un grand mur symbolisant la timeline est un prérequis.
2. Inviter les bonnes personnes (métier, UX, fonctionnel, technique)
3. Préparer des post-it aux bonnes couleurs (cf Déroulement) et des marqueurs (1 par personne)
4. Modéliser un ensemble de process métier cohérents à partir d'événements sur une timeline

Déroulement



Étape 1 : projeter les événements

Durant le premier round, tous les participants n'utilisent que les post-its orange, les événements du domaine. Ce sont des événements métier, qui viennent de se passer. Ils doivent donc être exprimés avec des verbes au passé



Event : Post-it orange, nom + verbe au passé

Événement clé du métier, a eu lieu et ne peut plus changer

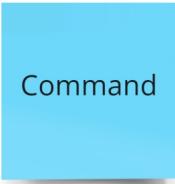
Positionnés sur la timeline, si possible dans un ordre chronologique

Étape 2 : ajouter les actions et les acteurs associés

On identifie maintenant les actions (commandes) à l'origine des événements déjà listés. Elles peuvent venir d'actions utilisateur, de systèmes externes ou encore d'actions/décisions internes. On y associe si besoin les acteurs qui lancent ces commandes. Une commande n'est pas forcément liée à un acteur

Command : Post-it bleu, verbe à l'infinitif + nom

Une **action sur le système**, représente une intention. Se produit en amont d'un événement



Command



Actor : Post-it jaune clair ≈ Persona

Personne à l'origine d'une action, désignée par son rôle, le plus précis possible.

Étape 3 : les traitements ou les règles métier



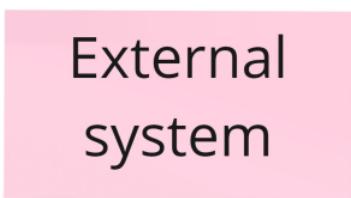
Policy

Policy : Post-it mauve, souvent une question

Traitement ou règle métier, conditionné, déclenché suite à un événement

Origine **Système** ou **Humain**

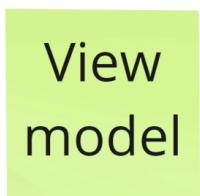
Étape 4 : les vues et les systèmes externes



External
system

External system : Post-it rose rectangulaire

Système à l'écoute des événements, brique purement technique ou composant métier en adhérence ne dépendant pas du périmètre métier modélisé

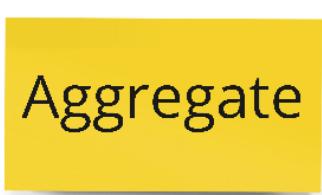


View
model

View Model (ou Read Model) : Post-it vert

Informations (en lecture) nécessaires pour prendre une décision, liées à l'interface utilisateur, qui permettent à l'acteur de prendre la décision d'exécuter une commande

Étape 5 : les agrégats



Aggregate

Aggregate : Post-it jaune moutarde rectangulaire, nom

Notion du métier sur laquelle s'appliquent les commandes, qui créent les événements

Leur nom se retrouve souvent dans des **commandes** et **events** assez proche.

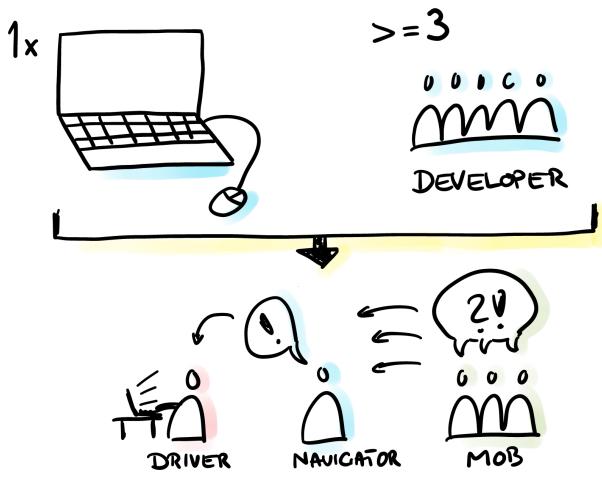
Ressources

- <https://www.eventstorming.com/>
- <https://www.agilepartner.net/en/eventstorming-from-big-picture-to-software-design/>
- [16. Glossaire métier partagé](#)
- <https://domaincentric.net/blog/modelling-aggregates-with-aggregate-design-canvas>
- <https://www.youtube.com/watch?v=Plo6jUl4VO8&t=2380s>
- https://martinfowler.com/bliki/DDD_Aggregate.html
- <https://cleandojo.com/2019/06/event-storming-modelisez-votre-domaine-metier-en-equipe/>

Mob Programming

- [Le principe](#)
- [Principaux bénéfices](#)
- [Rôles principaux](#)
 - [Le navigateur \(Navigator\) :](#)
 - [La foule \(mob\) :](#)
 - Rotation des rôles de conducteur et navigateur après un intervalle défini :
 - [Le facilitateur \(optionnel\) :](#)
 - [Le scribe :](#)
 - [Le chercheur en stackoverflow :](#)
- [Organisation \(remote\)](#)
 - [Au moins 3 personnes](#)
 - [Tout le monde en remote](#)
 - [1 écran avec le code toujours partagé](#)
 - [1 seul clavier actif](#)
 - [Des rotations courtes](#)
 - [Git à portée de main](#)
 - [Des décisions collectives](#)
 - [Une revue de code en fin de développement non nécessaire](#)
 - [On se fait confiance](#)
- [Dans la pratique](#)
 - [Mise en place d'un canal teams intégrant tous les membres de l'équipe](#)
 - [Outils](#)
- [Pour aller plus loin](#)

MOB PROGRAMMING IN A NUTSHELL



Le principe

"All the brilliant minds working together on the same thing, at the same time, in the same space, and at the same computer"
- Kevin Meadows

Une partie de l'équipe se réunit pour opter pour la résolution d'un problème défini à l'avance :

- Choisir un sujet
- Trouver les participants
- Démarrer la session

Principaux bénéfices

- S'assurer que l'on travaille sur des tâches réellement importantes
- Eliminer les changements de contextes ;
- Limiter le Bus Factor en partageant instantanément les connaissances (fonctionnelles, techniques)
- Accueillir de nouvelles personnes dans l'équipe
- Attaquer les problèmes au bazooka !

Autres rôles possibles :

Rôles principaux

Le conducteur (Driver) :

seule personne autorisée à écrire du code, il se laisse guider par les autres participants

Tips : commencer par un développeur les moins expérimenté sur le sujet traité permet de ne pas faire naître de stress inutile ("Je vais faire plus d'erreur", "Je connais moins bien le langage, cette librairie, ce pattern", "Il faut que j'aille aussi vite" ...)

Le navigateur (Navigator) :

donne des indications claires au conducteur en accord avec la foule

Tips : le navigateur doit jouer un rôle de protecteur du conducteur, en clarifiant les demandes du mob et en synthétisant les idées.

La foule (mob) :

réfléchit, discute et décrit les solutions à implémenter

L'audience (si trop de participants) :

écoute les échanges sans intervenir

Le facilitateur (optionnel) :

veille au respect des rôles, et incite à la participation

Le scribe :

assure la documentation des décisions prises lors de l'atelier (notes de réunion, ajout de tâches dans le backlog d'amélioration continue...)

on ne peut pas dire que c'est une bonne ou une mauvaise situation

Le chercheur en stackoverflow :

aide à trouver des solutions techniques aux problèmes rencontrés. Afin de fluidifier le déroulé de la session, il peut présenter les solutions envisagées lors d'une rotation entre 2 tours

Rotation des rôles de conducteur et navigateur après un intervalle défini :

- 1 membre du mob devient navigateur
- Le navigateur devient conducteur
- Le conducteur rejoint le mob

Organisation (remote)

Au moins 3 personnes

Pour les 3 rôle Mob / Navigateur / Conducteur

1 seul clavier actif

Le conducteur permet au reste de l'équipe de se concentrer sur la résolution du problème

Des décisions collectives

Les indications transmises au conducteur sont validées par l'équipe. Le code produit est conforme aux exigences de l'équipe

Tout le monde en remote

Éviter les échanges n'impliquant pas tous les participants

Des rotations courtes

Des sessions de 10 à 15 minutes pour garder la concentration active

Une revue de code en fin de développement non nécessaire

La revue de code est systématisée tout au long des développements

1 écran avec le code toujours partagé

Tous les participants sont concentrés sur le code produit

Git à portée de main

Le travail est effectué sur une branche récupérée par le conducteur à chaque rotation. On accepte que cette branche déclenche des builds KO

On se fait confiance

La communication avant, pendant et après l'exercice est primordiale pour

garder la cohésion, communiquer le résultat obtenu au reste de l'équipe

Dans la pratique

Mise en place d'un canal teams intégrant tous les membres de l'équipe

Facilitation de l'organisation d'une session :

- création d'une nouvelle réunion avec le sujet en titre
- appel des différents participants
- les autres membres de l'équipe peuvent se joindre à l'atelier plus facilement qu'avec des appels directs
- prise de notes possible pendant l'exercice pour assurer la communication avec les membres de l'équipe absents (notes de réunion)

Outils

<https://mobti.me/>

Pour aller plus loin

<https://github.com/willemlarsen/mobprogrammingrpg>

<https://mobprogramming.org/get-a-good-start-with-mob-programming/>

<https://mobprogramming.org/>

<https://proagileab.github.io/EnsembleEnablers/>

Mutation Testing (avec stryker.net)

- [What ?](#)
- [Objectifs](#)
- [Mise en place sur un projet](#)
 - [Création d'un fichier manifeste pour installer stryker au niveau de projet.](#)
 - [Installer stryker](#)
 - [Quand le reste de l'équipe récupère la solution, lancer la commande.](#)
 - [Configuration à mettre dans le projet de test](#)
- [Exécution](#)
- [Rapport de mutation](#)
- [Améliorer le score de mutation](#)

What ?

"**Mutation Testing** is a type of software testing in which certain statements of the source code are changed/mutated to check if the test cases are able to find errors in source code. The goal of Mutation Testing is ensuring the quality of test cases in terms of robustness that it should fail the mutated source code."

Objectifs

- détecter les portions de code non couvertes par des tests
- améliorer les pratiques de test

Mise en place sur un projet

Création d'un fichier manifeste pour installer stryker au niveau de projet.

```
dotnet new tool-manifest
```

Installer stryker

```
dotnet tool install dotnet-stryker
```

Quand le reste de l'équipe récupère la solution, lancer la commande.

```
dotnet tool restore
```

Configuration à mettre dans le projet de test

```
stryker-config.json
{
    "stryker-config": {
        "reporters": [
            "progress",
            "html"
        ],
        "ignore-methods": [ "ConfigureAwait" ]
    }
}
```

Documentation complète : <https://stryker-mutator.io/docs/stryker-net/configuration/>

Exécution

Dans une interface de terminal, depuis le dossier d'un projet de tests, lancer la commande

```
dotnet stryker
```

Pour afficher le rapport d'exécution au format HTML dans un navigateur, ajouter l'option -o :

```
dotnet stryker -o
```

Rapport de mutation

Une fois stryker exécuté, un rapport de mutation est généré au format HTML.

Chaque mutant dispose d'un statut :

- **Killed** : La mutation du code a fait échoué un test : le mutant a été détecté.
- **Ignored** : La mutation du code est dans un bloc de code exclus de la couverture de TU, le mutant est considéré comme détecté.
!\\ Un trop grand nombre de mutants en "Ignored" peut indiquer une utilisation abusive du "ExcludeFromCodeCoverage"
- **No Coverage** : La mutation est située dans un bloc de code non couvert par les TU : le mutant n'est pas détecté.
- **Survived** : La mutation est située dans un bloc couvert par des TU mais aucun n'a échoué : le mutant n'est pas détecté.

Le score de mutation correspond au pourcentage de mutants détectés sur l'ensemble des mutants générés avec un statut autre qu'ignored.

Un score de mutation > 80% est un bon résultat. Il signifie aussi que la couverture > 80% n'est pas qu'une couverture de façade et que les TU sécurisent la majeure partie du code.

Un score de mutation > 50 % & < 80% est un résultat mitigé.

Un score de mutation < 50% est un mauvais résultat. Sur un projet avec peu de couverture de code, il indique simplement un manque de tests. Sur un projet avec une forte couverture, il indique que cette dernière n'est que très peu sécurisante et laisse la place à un grand nombre de breaking changes non couverts.

Améliorer le score de mutation

- Les "**No Coverage**"
Le premier levier d'amélioration du score de mutation est d'agir sur les mutants flaggués "No Coverage". Pour ces derniers il suffit d'ajouter des TUs en faisant attention à bien inclure dans les assertions les cas remontés par les mutations.
Il est important de bien choisir ses assertions, dans le cas contraire les mutants risquent de persister avec un statut "Survived".
- Les "**Survived**"
Ici il est important d'améliorer les Tests Unitaires existants qui couvrent le code de chaque mutant. Une grande partie d'entre eux pourront sans doutes être traités en étant plus strict dans les assertions existantes ou en ajoutant de nouvelles assertions. De même préciser les arguments attendus lors de l'utilisation de mocks et vérifier les appels à la fin de l'exécution des tests améliore la robustesse des tests existants.
Il est aussi possible dans certains cas de devoir écrire de nouveaux tests spécifiques.
- Les "**Ignored**"
Si un nombre trop important d'ignored est généré, il est nécessaire de retirer les annotations "ExcludeFromCodeCoverage". Cette annotation doit se limiter aux classes techniques de bases (Startup & Program par exemple), éventuellement à certains morceaux de code générés ou à des classes d'infrastructure si celles-ci sont couvertes via d'autres types de tests (tests d'intégration par exemple).

Pousser des tests specflow dans XRay

Sources

- Documentation: [Importer les résultats de test Intégration dans Xray](#)
- Enregistrement du déroulement de la documentation: https://officediscount-my.sharepoint.com/:f/g/personal/nicolas_barlogis_cdbdx_biz/EIVJq6rvjAZJmeIngige48wBK_aNZW2YXbWZksw79r5KYA?e=oavw0H

Property-Based Testing (PBT)

- [C'est quoi le Property-Based Testing ?](#)
- [Objectifs](#)
- [Example Based vs Property-Based](#)
 - [Example-Based Testing](#)
 - [Property-Based Testing](#)
- [Si un test échoue, ça veut dire quoi ?](#)
 - [Shrinking](#)
- [Mise en place](#)
- [Exemple du Calculator](#)
 - [Identifier des propriétés](#)
 - [Utiliser l'attribut "PropertyAttribute"](#)
 - [Utiliser des Facts avec XUnit](#)
- [Comment on génère des objets "complexes" ?](#)
- [Use Cases \(les plus fréquents\)](#)
- [Anti-patterns associés](#)
- [Ressources](#)

C'est quoi le Property-Based Testing ?

L'idée est simple : identifier et tester des invariants.

1 invariant : quelque chose qui sera toujours vrai quelles que soient les données que vous fournissez à votre algorithme.

Pour cela, il faut utiliser un framework qui va générer des données aléatoires et vérifier si l'invariant reste vrai.

À chaque exécution de notre suite de tests, celui-ci va tester différentes combinaisons.

i Il est important de noter qu'un test de PBT en succès ne signifie pas que l'implémentation est correcte, il veut dire que le framework n'a pas su mettre en défaut l'implémentation.

Objectifs

- Identifier des Edge-cases auquel on a pas pensé (nulls, negative numbers, weird strings, ...)
- Avoir une meilleure compréhension métier : identifier les invariants métiers nécessitent une profonde compréhension métier

Example Based vs Property-Based

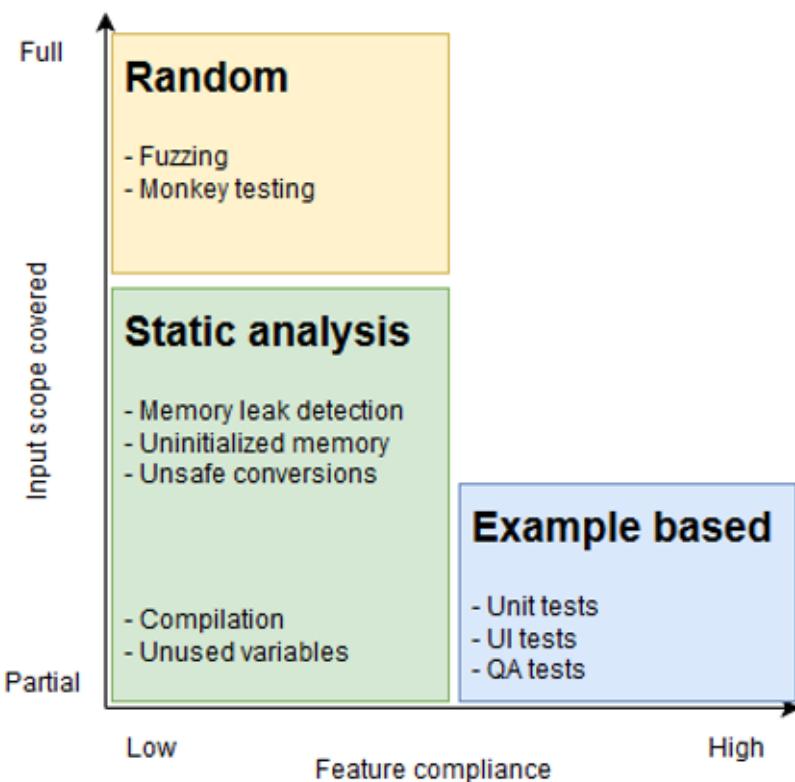
Example-Based Testing

En général lorsqu'on va écrire nos tests, nous allons nous focaliser sur des exemples et un scope d'inputs que nous avons identifié :

```

Given (x, y, ...) // Arrange
When I [call the subject under test] with (x, y, ...) // Act
Then I expect (output) // Assert

```



Ainsi nous validons que notre implémentation est valide avec ce qui était attendu (business requirements / critères d'acceptation) mais sur un scope de données réduit.

Property-Based Testing

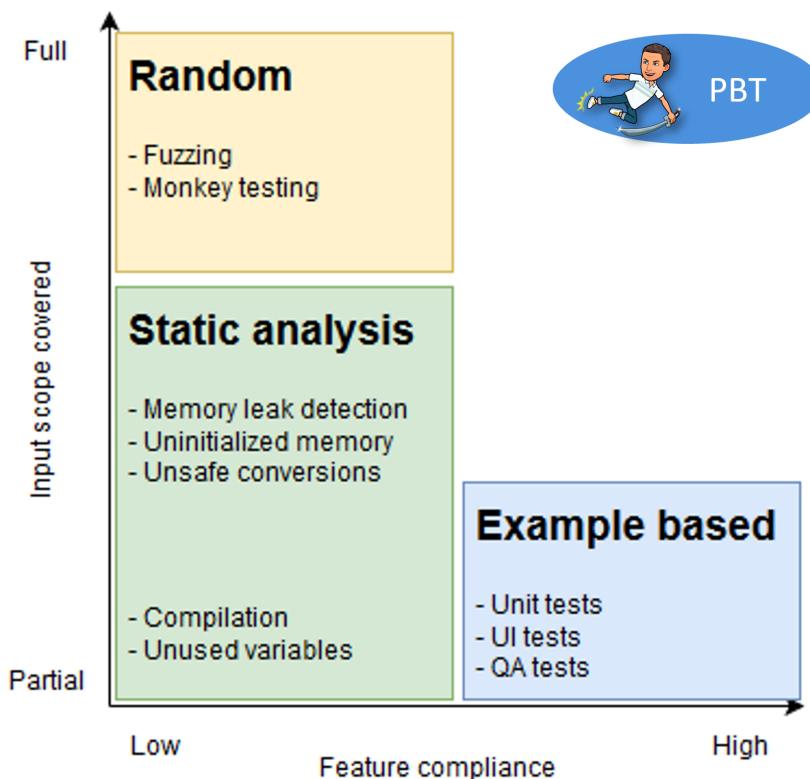
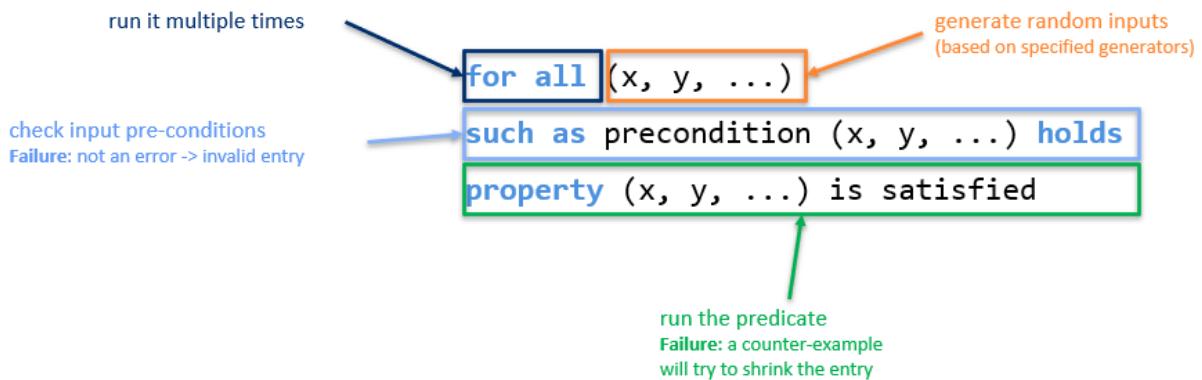
Avec PBT la promesse est de pouvoir vérifier que notre implémentation est valide du point de vue business mais avec un scope de données beaucoup plus important.

```

for all (x, y, ...)
such as precondition (x, y, ...) holds
property (x, y, ...) is satisfied

```

- Describe the input
- Describe the *properties* of the output
- Have the computer try lots of random examples
 - *Check if it fails*



Si un test échoue, ça veut dire quoi ?

Si le framework arrive à trouver un cas limite, il existe trois possibilités :

- le code de production n'est pas correct
- la façon dont l'invariant est testé n'est pas correcte
- la compréhension et définition de l'invariant ne sont pas correctes

Il est important d'avoir cette réflexion dès qu'un cas est identifié.

Quoi qu'il en soit, le framework est capable de vous donner les données utilisées pour mettre à mal votre code, vous pouvez donc facilement écrire un TU classique pour reproduire le cas.

Shrinking

Un bon framework de PBT est capable de faire du **shrinking**.

Une fois le cas limite identifié, celui-ci va travailler sur les données utilisées pour essayer de les simplifier au maximum tout en reproduisant l'erreur.

Ceci nous facilite l'effort d'analyse : imaginez une fonction qui prend une liste en argument, est-ce ma liste de 250 éléments ou juste un élément qui plante mon code ? S'il s'agit d'un élément, le shrinking peut l'isoler.

Mise en place

Les différentes librairies disponibles sur nos langages se basent sur QuickCheck.

En C# on va utiliser FsCheck + FsCheck.XUnit (pour l'intégration avec XUnit)

How to

```
dotnet add package FsCheck  
dotnet add package FsCheck.Xunit
```

A partir de là nous sommes prêt à implémenter nos premières propriétés.

Exemple du Calculator

Imaginons que nous voulons tester 1 Calculator :

Calculator

```
using System;  
namespace PBTKata.Math  
{  
    public static class Calculator  
    {  
        public static int Add(int x, int y) => x + y;  
    }  
}
```

En identifiant des exemples, nous pourrions écrire des TU comme suit :

Example tests

```
namespace PBTKata.Tests.Math.Solution  
{  
    public class CalculatorTests  
    {  
        private readonly Random random = new();  
        private readonly int times = 100;  
  
        private int RandomInt() => random.Next(int.MinValue, int.MaxValue);  
  
        [Fact]  
        public void Return4WhenIAdd1To3() => Add(1, 3).Should().Be(4);  
  
        [Fact]  
        public void Return2WhenIAddMinus1To3() => Add(-1, 3).Should().Be(2);  
  
        [Fact]  
        public void Return99WhenIAdd0To99() => Add(99, 0).Should().Be(99);  
    }  
}
```

Identifier des propriétés

On peut identifier facilement 3 propriétés mathématiques de l'addition :

- Commutativité
- Identité
- Associativité

Avec FsCheck on peut exprimer nos propriétés de 2 façons :

Utiliser l'attribut "PropertyAttribute"

Addition Properties

```
public class CalculatorProperties
{
    [Property]
    public Property Commutativity(int x, int y)
        => (Add(x, y) == Add(y, x)).ToProperty();

    [Property]
    public Property Associativity(int x)
        => (Add(Add(x, 1), 1) == Add(x, 2)).ToProperty();

    [Property]
    public Property Identity(int x)
        => (Add(x, 0) == x).ToProperty();
}
```

Ici on définit nos propriétés en créant des méthodes :

- Annotées avec l'attribut : **PropertyAttribute**
- Retournant une **Property**
 - FsCheck définit une méthode d'extension sur bool : **ToProperty()**

On peut collecter de la donnée sur les valeurs en inputs en utilisant la méthode **Collect**

Collect data

```
[Property]
public Property Commutativity(int x, int y)
    => (Add(x, y) == Add(y, x)).ToProperty().Collect($"x={x},y={y}");
```

Utiliser des Facts avec XUnit

Addition Properties with Facts

```
public class CalculatorPropertiesWithXUnitFact
{
    [Fact]
    public void Commutativity()
        => Prop.ForAll<int, int>((x, y) => Add(x, y) == Add(y, x))
            .QuickCheckThrowOnFailure();

    [Fact]
    public void Associativity()
        => Prop.ForAll<int>(x => Add(Add(x, 1), 1) == Add(x, 2))
            .QuickCheckThrowOnFailure();

    [Fact]
    public void Identity()
        => Prop.ForAll<int>(x => Add(x, 0) == x)
            .QuickCheckThrowOnFailure();
}
```

Ici on définit nos tests comme d'habitude avec XUnit :

- On utilise **Prop.ForAll** pour définir nos propriétés
- On les vérifie en appelant la méthode **QuickCheckThrowOnFailure()**
 - Celle-ci va lancer une exception si la propriété n'est pas satisfaite

Comment on génère des objets "complexes" ?

FsCheck définit des générateurs et shrinkers par défaut pour un grand nombre de types : bool, byte, int, float, char, string, DateTime, lists, array 1D/2D, Set, Map, objects.

FsCheck utilise la réflexion pour construire des record types, discriminated unions, tuples, enums et des classes "basiques" (utilisant que des types primitifs).

Si besoin on peut déclarer nos propres générateurs et les passer explicitement à nos propriétés :

Letter Generator static class

```
internal static class LetterGenerator
{
    public static Arbitrary<char> Generate() =>
        Arb.Default.Char().Filter(char.IsLetter);
}

// Quand on utilise l'attribut Property :
[Property(Arbitrary = new[] { typeof(LetterGenerator) })]
public Property Property(char c) => ...

// Quand on utilise Fact avec Prop.ForAll
[Fact]
public void Property()
    => Prop.ForAll(letterGenerator, ...)
        .QuickCheckThrowOnFailure();
```

Use Cases (les plus fréquents)

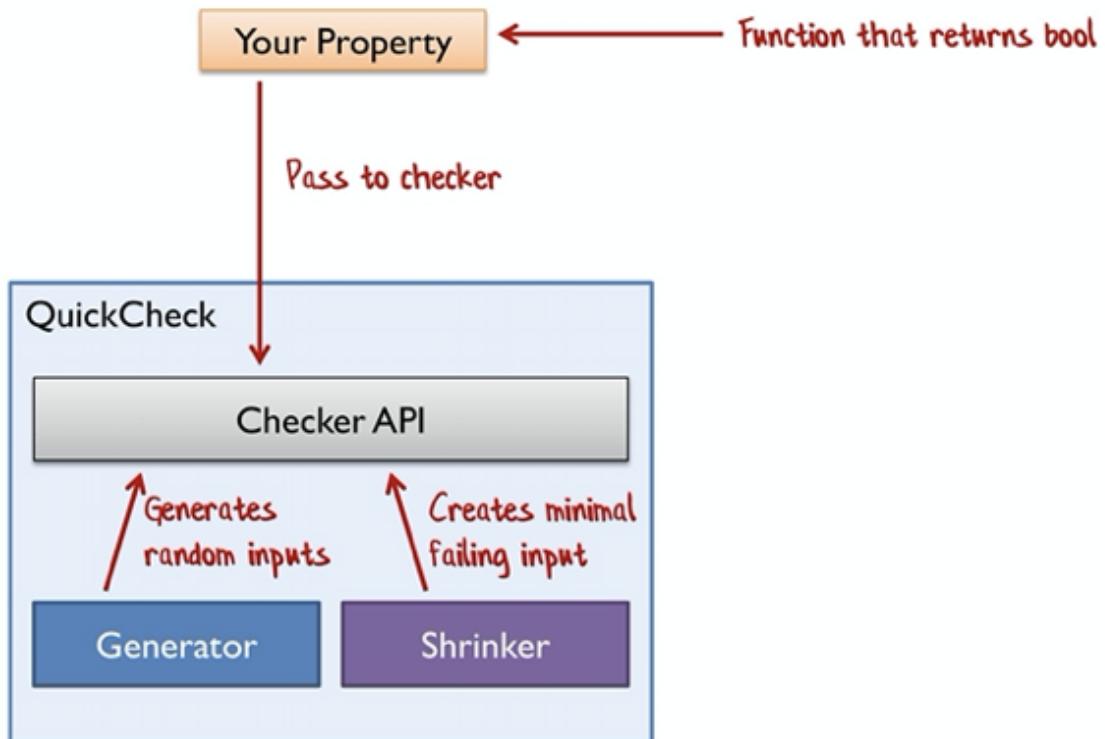
- Vérifier l'**idempotence**
 - $f(f(x)) == f(x)$
 - ex : UpperCase, Create / Delete
- Vérifier le **roundtripping**
 - $from(to(x)) == x$
 - ex : Serialization, PUT / GET, Reverse, Negate
- Valider des **invariants (propriétés universelles)**
 - $invariant(f(x)) == invariant(x)$
 - ex : Reverse, Map
- Vérifier la **commutativité**
 - $f(x, y) == f(y, x)$
 - ex : Addition, Min, Max
- Valider **ré-écriture**
 - $f(x) == new_f(x)$
 - Quand on réécrit ou optimise une implémentation
- Pour "upgrader" des tests paramétrisés
 - Remplacer les valeurs hardcodées / découvrir de nouveaux tests cases (edge cases)
- ...

Anti-patterns associés

- Quand un test échoue → relancer les tests
 - 2 exécutions d'un même test ne générera pas les mêmes inputs
 -

Au lieu de relancer → **investiguer sur la failure et ajouter 1 TU sur l'exemple identifié**

- Ré-implémenter le code de production
 - C'est souvent la dérive lorsqu'on écrit nos propriétés : avoir toute l'implémentation du code de production "leaker" dans les propriétés
 - Pour trouver des invariants, essayer de trouver des propriétés dites universelle sur votre code
- Filtrer les inputs de manière "ad-hoc"
 - Ne pas filtrer les valeurs d'entrée par nous-mêmes
 - Au lieu de cela, utilisez le support du framework (filtering, shrinking)



Ressources

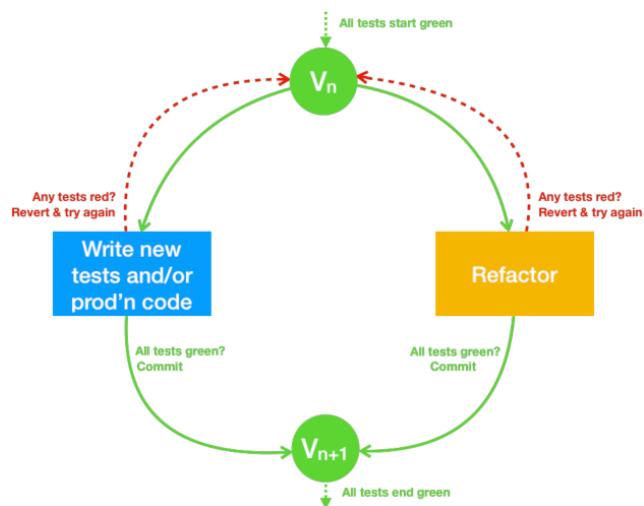
- [FsCheck documentation](#)
- [Intégration FsCheck avec XUnit](#)
- ["Functions for nothing, and your tests for free" Property-based testing and F# - George Pollard](#)
- [A la découverte du Property-Based Testing](#)
- [Solving the Diamond Kata with PBT in C#](#)
- [An introduction to PBT - Scott Wlaschin](#)
- [A journey to PBT - Katas](#)
- [Présentation interne du PBT par Julien Lafargue](#)

Test && Commit || Revert (TCR)

- [C'est quoi ?](#)
- [Objectifs](#)
- [Pourquoi ?](#)
- [Mise en place sur un projet](#)
 - [Utilisation de scripts](#)
 - [Utilisation d'un utilitaire \(En Go\)](#)
- [Ressources](#)

C'est quoi ?

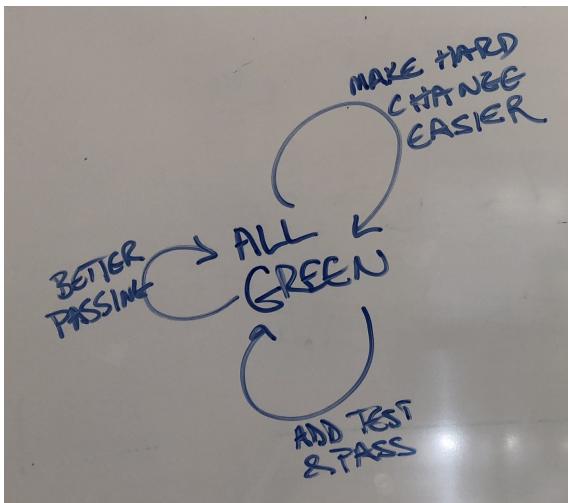
TCR est un workflow de développement comme décrit dans le diagramme ci-dessous :



- **Test && Commit || Revert :**
 - Test : lancer les tests
 - Commit : **si les tests sont verts** on commit
 - Revert : **si non** on revert

Objectifs

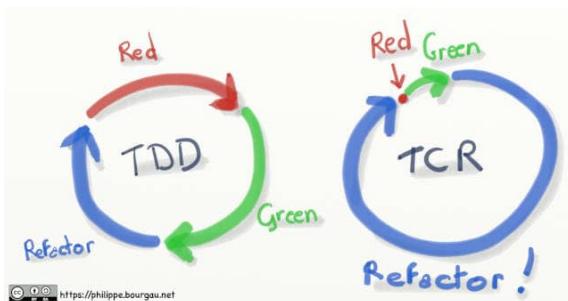
- Toujours garder tous les tests verts
- Facilitez les changements difficiles :
 - En travaillant en baby steps / micro-incréments
- Réduire le temps entre l'idée et la réussite d'un test d'une manière ou d'une autre.



Pourquoi ?

TCR impose le développement par petites étapes, en mettant l'accent sur :

- Le fait de toujours garder le feu vert pour les tests
- L'amélioration constante de notre code (refactoring)



Mise en place sur un projet

Parce que TCR est un workflow de développement on a plusieurs options :

- Implémenter le workflow à la main
- Utiliser des outils / scripts nous permettant de le mettre en oeuvre de façon automatique

Avant de vouloir utiliser TCR sur un projet existant, il est fortement recommandé de l'appréhender sur un ou plusieurs katas auparavant.

Utilisation de scripts

- Récupérer les scripts dédiés pour CSharp disponible ici : <https://github.com/Tr00d/TcrKata>
 - tcrw
 - folder tcr
- Les coller à la racine de votre repository
- Puis lancer le script :

Run tcr.exe

```
tcrw -p -t xunit
```

Utilisation d'un utilitaire (En Go)

Murex a développé un petit utilitaire en Go disponible pour Mac, Windows et Linux

- Pour l'installer, suivre cette documentation : <https://github.com/murex/TCR#running-tcr>
- Une fois installé, il suffit de lancer la commande :

Run tcr.exe

```
./tcr -b <base-directory> -l csharp -t dotnet
```

Vous devez préciser le répertoire dans lequel l'utilitaire va se mettre à l'écoute afin de détecter des changements sur le code de production et pouvoir lancer les tests

Ressources

- [Test && Commit || Revert expliqué par Kent Beck](#)
- [Does programming equal refactoring ? par Philippe Bourgau](#)
- [Murex TCR utility](#)
- [Tcr Kata par Guillaume Faas](#)

Test data builders

- [Objectifs](#)
- [Les 4 règles du Test Data Builder](#)
- [Exemple](#)
- [Générer automatiquement un squelette de builder](#)
- [Tips](#)
 - [Définir des états par défaut](#)
 - [Nommage des méthodes](#)
 - [Combiner les Tests Data Builders](#)
 - [Créer des objets ayant des similitudes](#)
 - [Générer des valeurs par défaut aléatoires à l'aide de fuzzers](#)
- [Références](#)
- [Training](#)

Objectifs

- Faciliter l'écriture des tests en permettant de créer facilement les inputs ou les outputs
- Faciliter la maintenabilité des tests en découplant la création des objets dans les tests et en l'isolant en un emplacement unique (SRP)
- Diffuser la compréhension fonctionnelle dans l'équipe par l'utilisation de termes métier
- Faciliter la lisibilité des tests en explicitant uniquement les données impliquées par le cas d'utilisation

Les 4 règles du Test Data Builder

1. Possède un variable d'instance pour chaque paramètre du constructeur ou chaque attribut public nécessaire.
2. Initialise ces variables d'instance avec des valeurs cohérentes, couramment utilisées et/ou valides.
3. Possède une méthode `Build()` responsable de la création du nouvel objet en utilisant les attributs de l'instance.
4. Possède des méthodes public "fluent" (ou chainables) permettant de modifier la valeur des attributs d'instance avant construction de l'objet.

Exemple

Objet à construire

```
public class CreatePropertyCommand : ICommand<Response>
{
    public string Label { get; set; }

    public bool? IsRanged { get; set; }

    public bool? IsNumeric { get; set; }

    public bool? IsMultiple { get; set; }

    public IEnumerable<string> Choices { get; set; }
}
```

Lors de l'utilisation dans les tests, il est important d'expliquer les données ayant un impact sur le comportement testé, même si cela revient à redéfinir une valeur par défaut avec la même valeur.

Test data builder

```
class CreatePropertyCommandBuilder
{
    private string _label;
    private bool _isMultiple = false;
    private bool _isRanged = false;
    private bool _isNumeric = false;
    private string[] _choices = new string[0];

    // Méthodes static de création d'un nouveau
    public static CreatePropertyCommandBuilder AMu
    {
        => new CreatePropertyCommandBuilder { _ }
    }

    public static CreatePropertyCommandBuilder .
    => new CreatePropertyCommandBuilder { _ }

    public static CreatePropertyCommandBuilder .
    => new CreatePropertyCommandBuilder { _ }

    public static CreatePropertyCommandBuilder .
    => new CreatePropertyCommandBuilder { _ }

    // Méthodes fluent permettant de modifier l
}
```

```

Utilisation dans les tests
[Fact]
public async Task Should_create_a_property_with_valid_body()
{
    //Arrange
    CreatePropertyCommand givenCommand = CreatePropertyCommandBuilder.WithChoices(
        .WithChoices(new string[] { "1", "2" })
        .LabeledBy("Ma nouvelle propriété")
        .Build());
}

// ...
// Assert
property.Label.Should().Be("Ma nouvelle propriété");
property.Choices.Should().BeEquivalentTo(new string[] { "1", "2" });
}

public CreatePropertyCommandBuilder Labeled(
    string label)
{
    _label = label;
    return this;
}

public CreatePropertyCommandBuilder WithChoices(
    string[] choices)
{
    _choices = choices;
    return this;
}

// Méthode de construction de l'objet cible
public CreatePropertyCommand Build()
{
    return new CreatePropertyCommand(
        Choices = _choices,
        Label = _label,
        IsMultiple = _isMultiple,
        IsRanged = _isRanged,
        IsNumeric = _isNumeric
    );
}
}

```

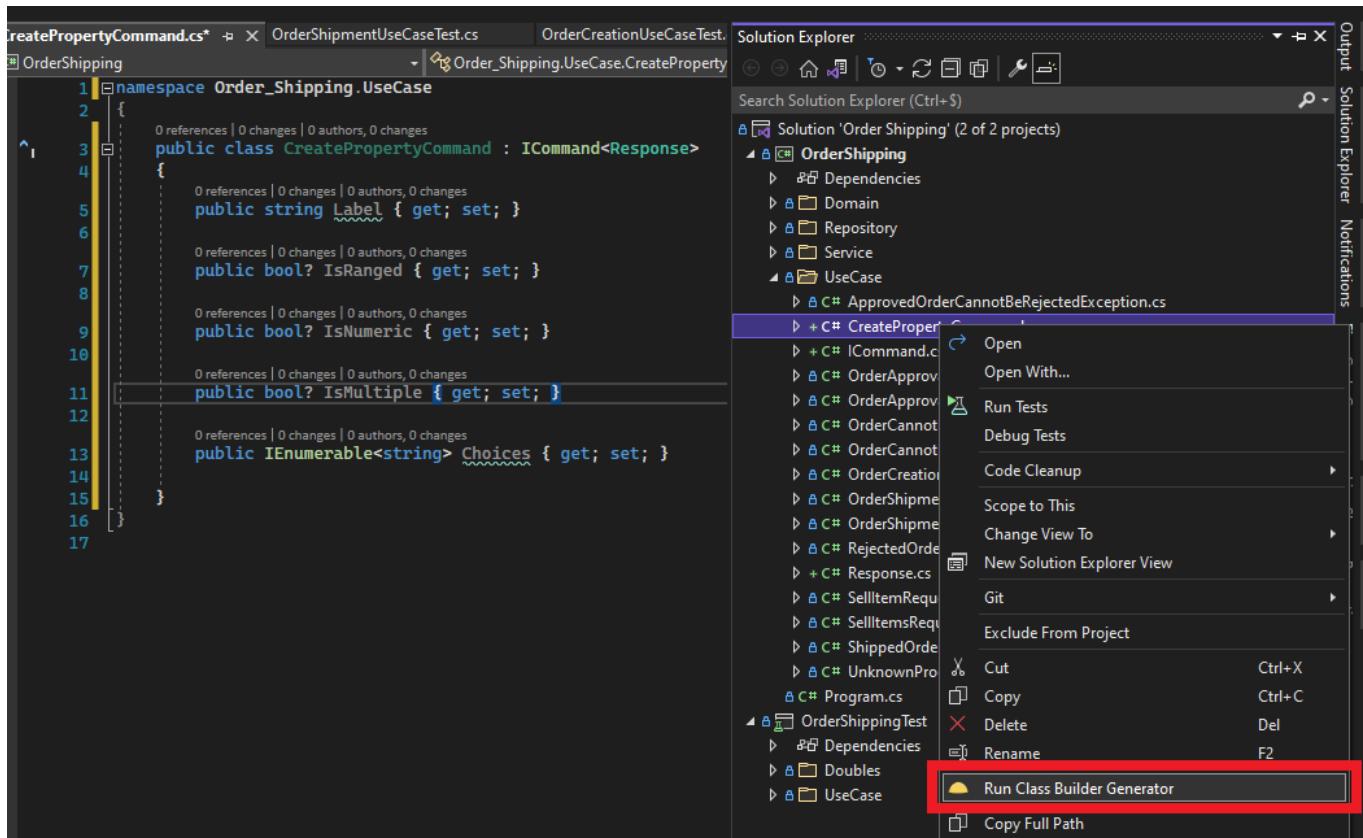
Générer automatiquement un squelette de builder

Pour faciliter l'écriture d'un builder, il est possible de générer un builder générique puis d'y apporter de la personnalisation.

L'extension [C# model to builder](#) fonctionne également très bien (à condition que la classe soit comment par "public class") et est disponible sur Visual Studio et Visual Studio Code.

Il existe de nombreux plugins permettant de générer différentes formes du pattern builder pour vos classes, nous allons ici juste voir un exemple avec l'extension [Class Builder Generator](#) disponible sous Visual Studio.

Une fois cette extension installée, on peut simplement faire un clic droit sur une de nos classe et utiliser le bouton "Run Class Builder Generator"



La classe `CreatePropertyCommandBuilder` est alors générée. Cette méthode permet de gagner du temps dans la création d'un builder générique, auquel on peut alors ajouter des méthodes orientées métier ou des constructeurs statiques.

```
public class CreatePropertyCommandBuilder
{
    private string label;
    private bool? isRanged;
    private bool? isNumeric;
    private bool? isMultiple;
    private IEnumerable<string> choices;

    /// <summary>
    /// Create a new instance for the <see cref="CreatePropertyCommandBuilder">CreatePropertyCommandBuilder
    /// </summary>
    public CreatePropertyCommandBuilder()
    {
        Reset();
    }

    /// <summary>
    /// Reset all properties' to the default value
    /// </summary>
    /// <returns>Returns the <see cref="CreatePropertyCommandBuilder">CreatePropertyCommandBuilder</returns>
    public CreatePropertyCommandBuilder Reset()
    {
        label = default;
        isRanged = default;
        isNumeric = default;
        isMultiple = default;
        choices = default;

        return this;
    }

    /// <summary>
```

```

/// Set a value of type <typeparamref name="string"/> for the property <paramref name="label">
/// <summary>
/// <param name="label">A value of type <typeparamref name="string"/> will be defined for the
/// <returns>Returns the <see cref="CreatePropertyCommandBuilder">CreatePropertyCommandBuilder</see>
public CreatePropertyCommandBuilder WithLabel(string label)
{
    this.label = label;
    return this;
}

// ..... //



/// <summary>
/// Build a class of type <see cref="CreatePropertyCommand">CreatePropertyCommand</see> with a
/// <summary>
/// <returns>Returns a <see cref="CreatePropertyCommand">CreatePropertyCommand</see> class</returns>
public CreatePropertyCommand Build()
{
    return new CreatePropertyCommand
    {
        Label = label,
        IsRanged = isRanged,
        IsNumeric = isNumeric,
        IsMultiple = isMultiple,
        Choices = choices,
    };
}
}

```

Tips

Définir des états par défaut

Combiner le pattern Test Data Builder avec le pattern Object Mother permet de simplifier l'écriture de cas de test récurrents

```

// Méthode de construction générique
var category = CategoryBuilder.ACategory.WithLevel(3)
    .ReferencedBy("AABBCC")
    .WithParent("AABB")
    .WithParentReferences(new[] { "AA", "AABB" })
    .Build();

// Pattern Object Mother
var category = CategoryBuilder.ALevel3Category
    .ReferencedBy("AABBCC")
    .WithParent("AABB")
    .WithParentReferences(new[] { "AA", "AABB" })
    .Build();

```

De même le respect des invariants métier peut être simplifié lors de l'implémentation des méthodes de modification :

```

// CategoryBuilder.cs
public CategoryBuilder ReferencedBy(string reference)
{
    var parent = reference.Length > 2 ? reference.Substring(0, reference.Length - 2) : string.Empty;
    var grandParent = parent.Length > 2 ? parent.Substring(0, reference.Length - 2) : string.Empty;
    var parentReferences = new List<string> { grandParent, parent }.Where(s => s != string.Empty).ToList();

    _reference = reference;
    _id = reference;
    _parent = parent;
    _label = $"Category {_reference}";
    _parentReferences = parentReferences;
    return this;
}

// CategoryTests.cs

```

```

var aLevel3Category = CategoryBuilder.ALevel3Category
    .ReferencedBy( "AABBCC" )
    .Build();

var aLevel1Category = CategoryBuilder.ALevel1Category
    .ReferencedBy( "AA" )
    .Build();

```

Nommage des méthodes

Ne pas se limiter à un pattern de nommage générique (ex: `with[Property](. . .)`) permet une meilleure lisibilité des tests et explicite l'intention et le comportement applicatif testé :

```

// Nommage générique des méthodes :
var invoice = InvoiceBuilder.AnInvoice().WithCustomer(customer).WithDeliveryAddress(address).Bu
// Intention explicitée :
var invoice = InvoiceBuilder.AnInvoice().For(customer).DeliveredAt(address).Build();

```

Combiner les Tests Data Builders

Lorsque la construction d'un objet est effectuée à l'aide d'un Test Data Builder fait appel à des objets créés par des Tests Data Builders, utiliser les builders en tant qu'argument des méthodes de modification d'état permet de réduire le bruit et d'obtenir des tests plus lisibles :

```

// Avant
var invoice = InvoiceBuilder.AnInvoice()
    .For(CustomerBuilder.ACustomer()
        .Named("John Doe")
        .Build())
    .DeliveredAt(AddressBuilder.AnAddress()
        .WithNoPostCode()
        .Build())
    .Build();

// Après
var invoice = InvoiceBuilder.AnInvoice()
    .For(CustomerBuilder.ACustomer()
        .Named("John Doe"))
    .DeliveredAt(AddressBuilder.AnAddress()
        .WithNoPostCode())
    .Build();

```

Dans ce cas, la construction des dépendances nécessitant l'utilisation de builders est déléguée dans la méthode `Build()` du parent.

Créer des objets ayant des similitudes

L'utilisation de Test Data Builders permet de créer des instances ayant de fortes similitudes de manière "clean", en évitant l'introduction de duplication.

Supposons que nous voulions créer des instances d'une classe `Country` pour la France et l'Allemagne. Les 2 pays appartenant à la zone Euro, nous pourrions écrire le code suivant :

```

// CountryBuilder.cs

private CountryBuilder(CountryBuilder copy) {
    this._name = copy._name;
    this._continent = copy._continent;
    this._language = copy._language;
    this._currency = copy._currency;
}

public CountryBuilder But() {
    return new CountryBuilder(this);
}

// Tests.cs

```

```
[Fact]
public void Should()
{
    var aEuropeanCountry = CountryBuilder.ACountry().WithCurrency("EUR").OnContinent("Europe");
    var france = aEuropeanCountry.But().Named("France").WithLanguage("French").Build();
    var germany = aEuropeanCountry.But().Named("Germany").WithLanguage("German").Build();
}
```

Générer des valeurs par défaut aléatoires à l'aide de fuzzers

TODO : explication

```
// BrandBuilder.cs
public static BrandBuilder AnActiveBrand(IFuzz fuzz)
{
    string reference = fuzz.GeneratePositiveInteger(99999).ToString("00000");
    DateTime createdAt = fuzz.GenerateDateTimeBetween(DateTime.Today.Subtract(new TimeSpan(36
    return new BrandBuilder
    {
        _reference = reference,
        _id = reference,
        _name = fuzz.GenerateSentence(3),
        _state = BrandState.Active,
        _spellings = fuzz.GenerateWords(4) as string[],
        _createdAt = createdAt,
        _updatedAt = fuzz.GenerateDateTimeBetween(createdAt, DateTime.Now),
        _comment = fuzz.GenerateParagraph(),
        _guid = fuzz.GenerateGuid().ToString(),
    };
}

// FuzzerExtension.cs
public static class FuzzerExtension
{
    public static BrandBuilder AnActiveBrand(this IFuzz fuzz)
    {
        return BrandBuilder.AnActiveBrand(fuzz);
    }
}

// BrandTests.cs
public class BrandTests
{
    private Fuzzer _fuzzer;

    public FuzzingActionResultTests(ITestOutputHelper testOutputHelper)
    {
        Fuzzer.Log = s => testOutputHelper.WriteLine(s);
        _fuzzer = new Fuzzer();
    }

    [Fact]
    public void Should_return_action_result_with_default_random_values()
    {
        var brand = _fuzzer.AnActiveBrand().LabeledBy("My brand name").Build();
        // ...
    }
}
```

Références

- <http://www.growing-object-oriented-software.com/>
- <https://blog.ploeh.dk/2017/08/15/test-data-builders-in-c/>
- <http://www.natpryce.com/articles/000714.html>
- <https://www.arhohuttunen.com/test-data-builders/>

Training

- http://tfs.cdbdx.biz:8080/tfs/defaultcollection/DotNet-TrainingCourses/_git/DotNetCore-TrainingExercises

Tests d'Architecture avec ArchUnit

- [Qu'est-ce qu'ArchUnit ?](#)
- [Pourquoi est-il intéressant de faire des tests d'architecture sur nos projets ?](#)
- [Quelques exemples d'usage !](#)
- [Tutoriel de mise en place concrète en Dotnet C#](#)

Qu'est-ce qu'ArchUnit ?

ArchUnit est une librairie qui permet de faire des tests automatisés d'architecture, afin de vérifier que l'application suit bien les règles d'architecture mise en place sur le projet. Cette librairie est disponible pour plusieurs langages, notamment C# (ArchUnit.NET) et Java (ArchUnit).

Pourquoi est-il intéressant de faire des tests d'architecture sur nos projets ?

L'architecture logicielle peut-être définie selon Martin Fowler, comme un ensemble de décisions importantes et sur lesquelles il est difficile de revenir après coup.

Il peut être donc intéressant de laisser une trace dans le code de la décision prise et des tests pour s'assurer qu'elle est toujours effective dans le temps dans le code base, cela de manière automatique.

L'automatisation sert de fillet de sécurité en plus de documentation pour tous développeurs souhaitant modifier le code en respectant les conventions d'architecture mises en place.

Cela peut servir aussi à l'onboarding de nouveau développeur qui ne maîtrisent/connaissent pas forcément les principes d'architecture en question.

D'autres pratiques comme les ADRs (Architecture Décision Records) peuvent faire office de documentation, mais vous n'aurez pas le côté automatisé apporté par Arch Unit.

Quelques exemples d'usage !

Le cas d'utilisation assez classique est de contrôler que le style d'architecture choisi soit bien respecté, par exemple:

- Dans une architecture n-tiers: controller -> services -> persistence
- Dans une architecture hexagonale: controller -> domain <- persistence

Lire -> comme dépend de ! En gros ce qui va nous intéresser dans ses tests c'est de vérifier le sens des dépendances

Par exemple dans le cadre de l'architecture hexagonale, que le domaine ne dépend pas de technologie autre que le langage choisi (Voir quelques librairies triées sur le volet pour éviter de parfois réinventer la roue dans certains domaines)

Autre cas d'utilisation, les conventions de nommage, par exemple si vous avez fait le choix d'utiliser le pattern CQRS, il est courant d'avoir des Events, Commands et des Handler. Il peut être intéressants d'enforcer le nommage des événements déclarer dans des packages spécifiques, histoire de voir rapidement dans le projet, l'ensemble des événements pouvant être générés et commandes interceptées.

Il y a évidemment plein d'autres cas d'utilisation, pour lesquelles nous ne rentrerons pas dans les détails:

- S'assurer des retours de certaines méthodes (ex: contrôleur return des Api Response, CommandHandler return void)
- S'assurer que chose contenant des termes spécifiques soit dans les bons packages (ex: contrôleur)
- S'assurer que les méthodes contenant "Is" ou "Has" retournent bien des Boolean

- S'assurer que les méthodes "Get" getter retourne toujours quelque chose
- S'assurer qu'il n'y a pas de dépendance circulaire dans nos services

Tutoriel de mise en place concrète en Dotnet C#

Installation d'ArchUnit depuis Nuget

```
dotnet add package TngTech.ArchUnitNET --version 0.9.1
```

Vous pouvez choisir une version précise si besoin

```
dotnet add package TngTech.ArchUnitNET --version 0.9.1
```

Installer l'extension selon l'outil de test que vous utilisez sur votre projet

```
dotnet add package TngTech.ArchUnitNET.xUnit --version 0.9.1
dotnet add package TngTech.ArchUnitNET.NUnit --version 0.9.1
dotnet add package TngTech.ArchUnitNET.MSTestV2 --version 0.9.1
```

Créer une classe de test (Exemple avec xUnit)

```
namespace Architecture.Tests
{
    Public class HexagonalArchitectureTests
    {
        [Fact]
        public void ClassesInDomainCanOnlyAccessClassesInDomainItself() =>
            Classes().That()
                .ResideInNamespace( "Domain" )
                .Should()
                .OnlyDependOnTypesThat()
                .ResideInNamespace( "Domain" )
                .Check();
    }
}
```

Pour aller plus loin et voir d'autres exemples concrets:

- <https://github.com/TNG/ArchUnitNET/tree/main/ExampleTest>
- <https://github.com/ythirion/archunit-examples/tree/main/c%23/ArchUnit-demo-tests>