

# Les tests unitaires et tests de charge

## Test unitaires : aujourd'hui et demain



# Test unitaires : aujourd’hui et demain

## Plan de cette présentation

- Rappel : les types de tests
- Test unitaire & TDD
- BDD (*Behavior Driven Development*)

# Rappel : les types de tests



# Les catégories de tests logiciel

- Les tests fonctionnels
  - Les tests non fonctionnels
  - Maintenance
- 
- *Il faut noter que ces types ne sont pas applicables à tous les projets.*
  - *Cela dépend souvent de la nature et de la portée de ce dernier.*



# Tests Fonctionnels

- On parle de tests fonctionnels quand il s'agit de vérifier qu'une classe permet bien de remplir avec succès l'objectif fixé par un cas d'utilisation donné.
- Un test fonctionnel permet de répondre à la question « est-ce que le code permet de faire ça ? » ou « est-ce que cette fonctionnalité attendue est bien fonctionnelle ?



# Tests Non-Fonctionnels

- Les tests non-fonctionnels vérifient des propriétés qui ne sont pas directement liées à une utilisation du code.
- Il s'agit de vérifier des caractéristiques telle que la sécurité ou la capacité à monter en charge.
- Les tests non-fonctionnels permettent plutôt de répondre à des questions telles que « est-ce que cette classe peut être utilisée par 1000 threads en même temps sans erreur ? ».

# Maintenance

- Selon la définition de l'AFNOR, la maintenance vise à maintenir ou à rétablir un bien dans un état spécifié afin que celui-ci soit en mesure d'assurer un service déterminé.
- La maintenance regroupe ainsi les actions de dépannage et de réparation, de réglage, de révision, de contrôle et de vérification des logiciels.



# Les types de test logiciel

Categories	Type
Test fonctionnels	<u>Unit Testing</u> - <u>Integration Testing</u>
Test non fonctionnels	Performance - Endurance Load - Volume
Maintenance	Regression - Maintenance

# Test automatisés

- Un test automatisé est un test dont l'exécution ne nécessite pas l'intervention d'un humain.
- L'exécution de tests automatisés requiert donc l'utilisation de solutions informatiques dont le but est d'exécuter des actions, soit spécifiquement dans un navigateur web, soit plus généralement au niveau du système d'exploitation.
- Avantage : ces scripts peuvent être réutilisés pour tester votre produit ultérieurement.
- Inconvénient : cela implique un temps et un coût de création.

# Test manuels

- Lors de tests manuels, c'est une personne, un testeur expérimenté, qui va naviguer dans votre produit.
- Il l'utilise comme le feront vos futurs utilisateurs. Les tests manuels peuvent se dérouler de deux façons différentes :
- **Test avec scénario** - Le testeur suit des parcours définis au préalable pour contrôler la qualité de l'application sur des problématiques bien précises.
- **Test exploratoire** - Le testeur navigue librement dans l'application pour y déceler le maximum de bugs et de problèmes.

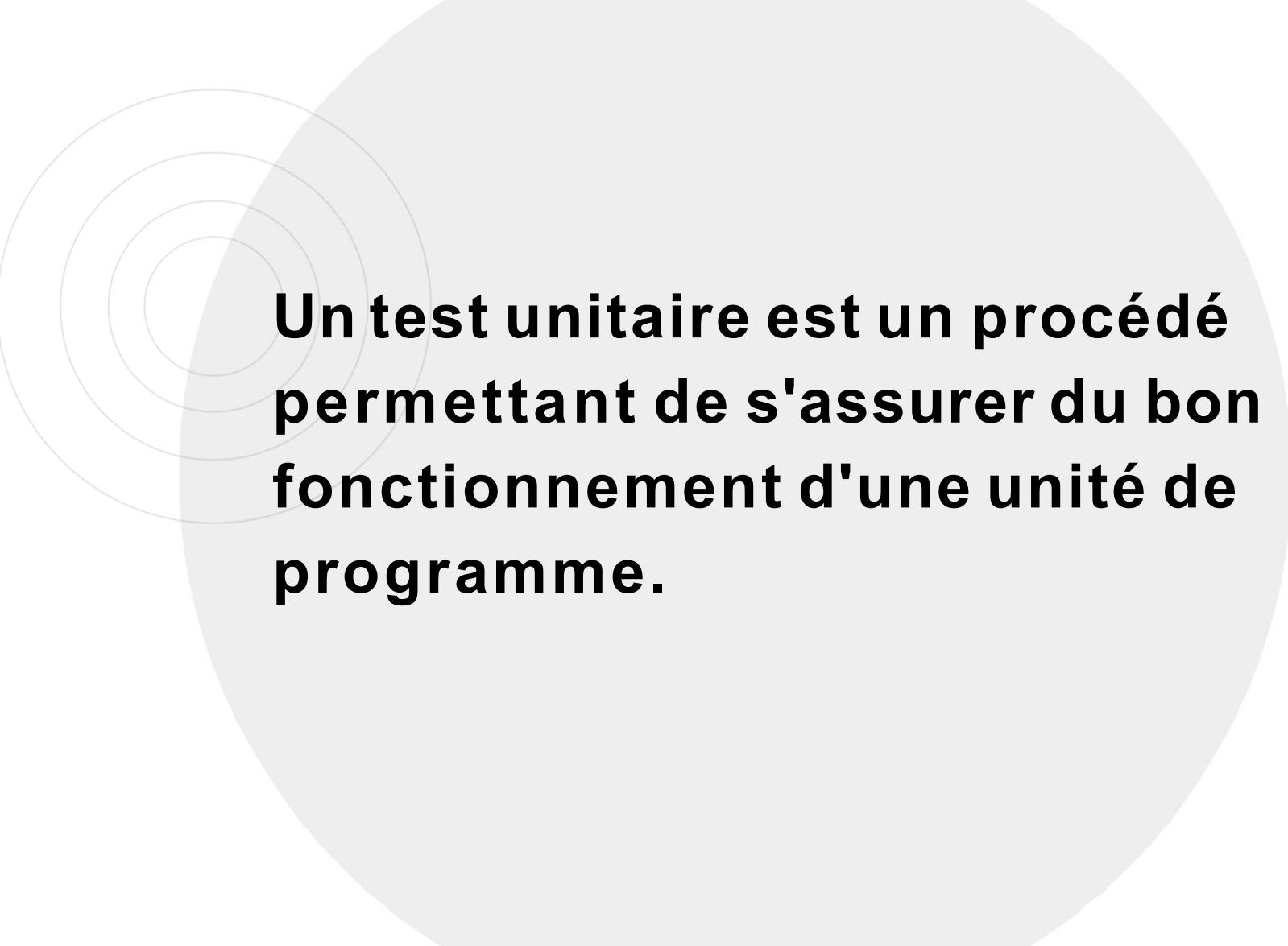
# Test manuels

- Avantage : Contrairement au test automatisé, le test manuel vous permet de tester l'UI (user interface) et UX (user experience) de votre produit : l'affichage correct du texte, les liens, les images.  
Cela vous permet de déceler des bugs qui seraient visibles par vos utilisateurs mais n'auraient pas pu être repérés par un robot.
- Inconvénient : Les tests manuels ne peuvent pas être répliqués aussi facilement que les tests automatisés.

# Test unitaire

**BUG HUNTER**





**Un test unitaire est un procédé permettant de s'assurer du bon fonctionnement d'une unité de programme.**



# Définition

Il s'agit simplement de vérifier, en fonction de certaines données fournies en entrée d'un module de code , que les données qui en sortent ou les actions qui en découlent sont conformes aux spécifications du module.



# Utilité

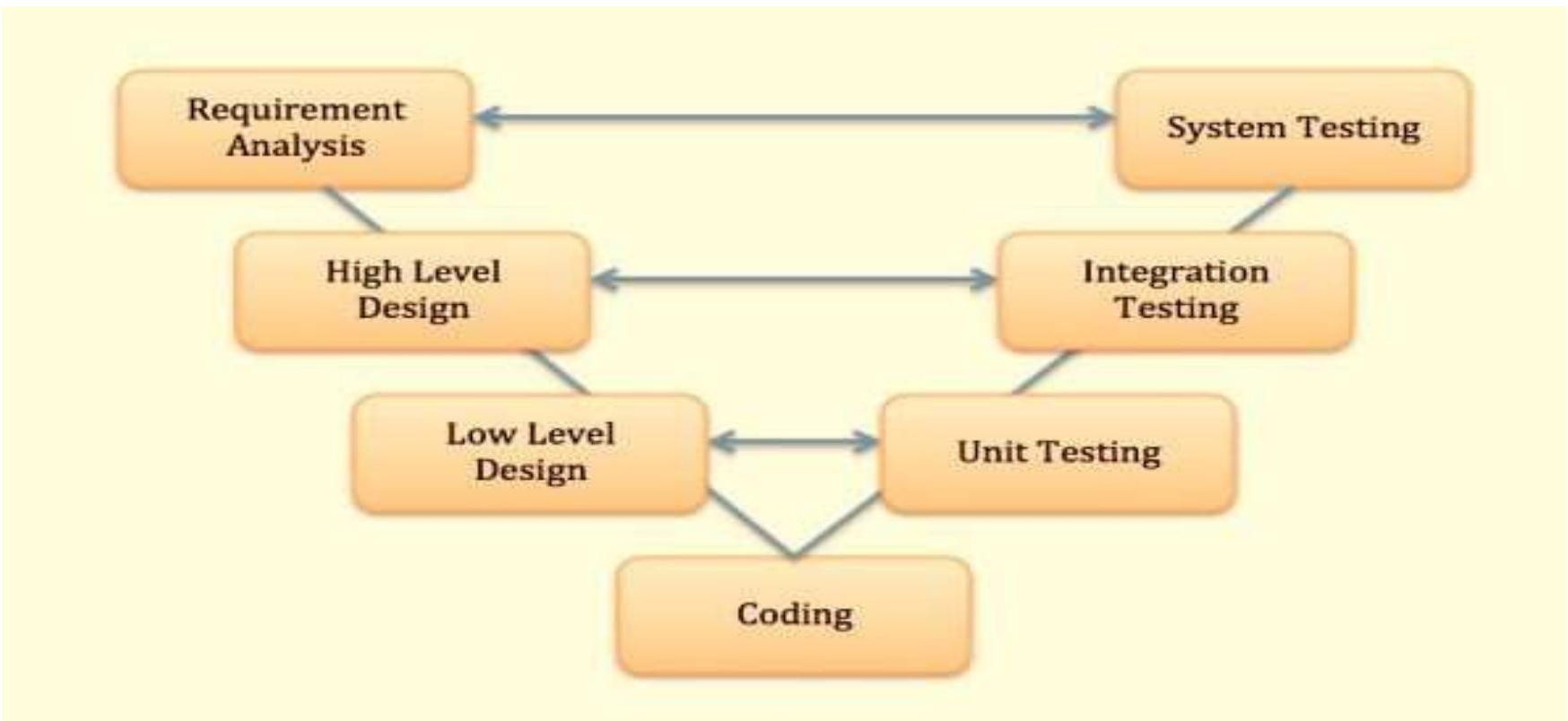
- Le test définit un critère d'arrêt (état ou sorties à l'issue de l'exécution) et permet de statuer sur le succès ou sur l'échec d'une vérification.
- Grâce à la spécification, on est en mesure de faire correspondre un état d'entrée donné à un résultat ou à une sortie.
- Le test permet de vérifier que la relation d'entrée / sortie donnée par la spécification est bel et bien réalisée.

# Utilité

- Trouver les erreurs rapidement
- Sécuriser la maintenance
- Documenter le code :
  - Il est possible que la documentation ne soit plus à jour, mais les tests eux correspondent à la réalité de l'application.



# Quand tester ?



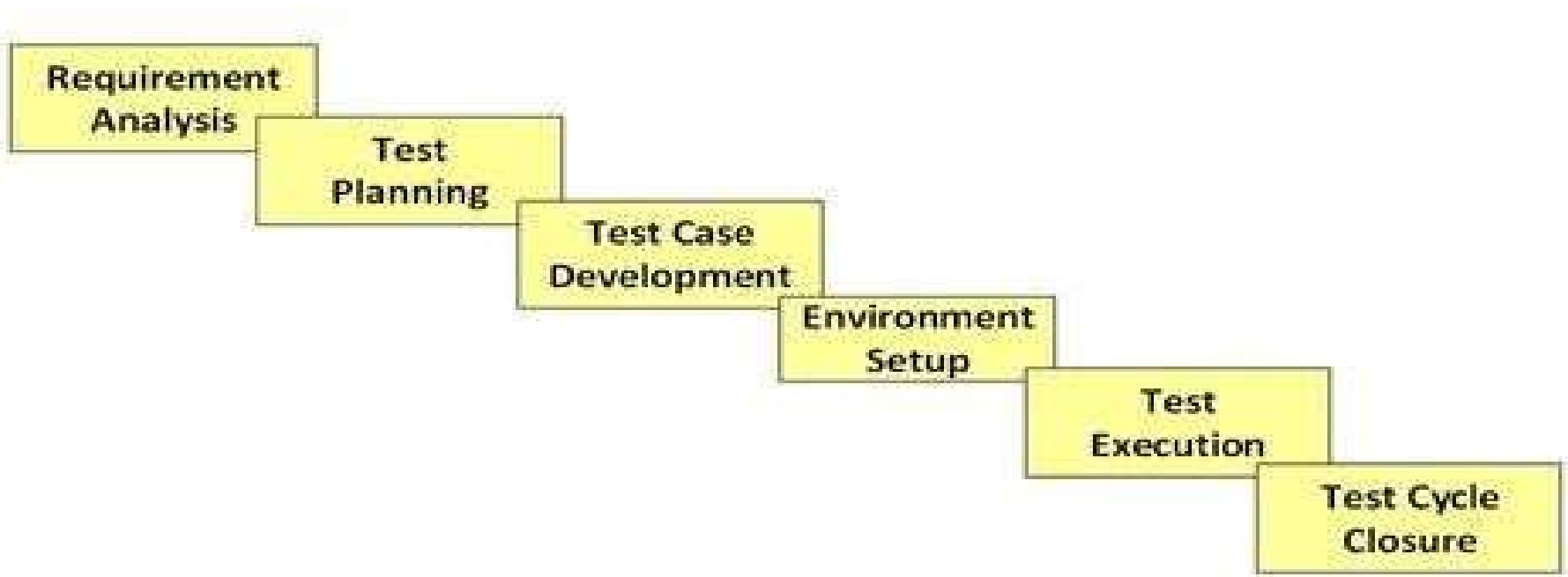
# Fonctionnement

On définit généralement 4 phases dans l'exécution d'un test unitaire :

1. **Initialisation** : définition d'un environnement de test complètement reproductible (une *fixture*).
2. **Exercice** : le module à tester est exécuté.
3. **Vérification** (utilisation de fonctions *assert*) : comparaison des résultats obtenus avec un vecteur de résultat défini. Ces tests définissent le résultat du test : SUCCÈS (SUCCÈS) ou ÉCHEC (FAILURE).
4. **Désactivation** : désinstallation des fixtures pour retrouver l'état initial du système, dans le but de ne pas polluer les tests suivants.

Tous les tests doivent être indépendants et reproductibles unitairement (quand exécutés seuls).

# Cycle de vie d'un test



# Utilisation des *mocks* (bouchons)

- Les **mocks** sont des objets permettant de simuler un objet réel de façon contrôlée.
- Dans certains cas, l'utilisation de **mock** est primordiale, pour un gain de temps de couverture de code, et de fiabilité des tests :
  - pour simuler une base de données, un service web, etc., les interactions entre l'application et ces outils prennent du temps, l'utilisation de **mock** pour simuler leurs fonctionnements peut être un gain de temps considérable ;
  - certains cas d'erreurs sont très difficile à reproduire, l'utilisation de **mock** permet ici de simuler une erreur pour pouvoir traiter ce cas et donc améliorer la couverture de code, par exemple le catch d'une exception ;
  - sans l'utilisation de **mock**, le test peut retourner une erreur ne provenant pas du code qui est testé (par exemple une base de données).

# Outils de Test Unitaire

- ❖ [AUnit11 pour Ada](#) ;
- ❖ [ASUnit12 pour ActionScript](#) ;
- ❖ [Cppunit13 pour C++](#) ;
- ❖ [CUnit14 pour C](#) ;
- ❖ [DUnit pour Delphi](#) ;
- ❖ [FLEXunit pour Adobe Flex](#) ;
- ❖ [Google Test15 et Boost Test16 pour C++](#) ;
- ❖ [HUnit17 pour Haskell](#) ;
- ❖ [JUnit18, QUnit19 et Unit.js20 pour JavaScript](#)
- ❖ [Tape pour JavaScript](#) ;
- ❖ [Test::Unit pour Ruby](#) ;
- ❖ [Test::More pour Perl](#) ;
- ❖ [JUnit21 et TestNG22 pour Java](#) ;
- ❖ [NUnit23 pour .NET](#) ;
- ❖ [Microsoft Unit Test24 pour .NET](#) ;
- ❖ [xUnit25 pour .NET](#) ;
- ❖ [NUnitASP26 pour ASP.NET](#)
- ❖ [OUnit27 pour OCaml](#) ;
- ❖ [OCunit pour Objective C](#) ;
- ❖ [PBUnit pour PowerBuilder](#) ;
- ❖ [PHPUnit28, SimpleTest29 et Atoum30 pour PHP](#) ;
- ❖ [utPLSQL 31 pour PL/SQL](#) ;
- ❖ [Unittest et PyUnit pour Python](#) ;

# Avantage, limites et best practices



# Avantages

- Les tests unitaires permettent notamment de documenter automatiquement le code source. En effet, la lecture des tests permet de comprendre le comportement attendu du code, et donc le fonctionnement de l'application, tant technique que logique.
- Les développeurs peuvent appréhender toutes les fonctionnalités apporté par un système.
- Nous pouvons tester des parties du projet sans attendre que d'autres soient terminées.
- Les tests automatisés représentent donc un formidable outil de non-régression.

# Limites

- Les tests unitaires ne détectent pas toutes les erreurs.
- Les tests unitaires ne peuvent pas évaluer tous les chemins d'exécution,
- Les tests unitaires sont centrés sur une unité de code, il ne peuvent pas détecter les erreurs d'intégration.

# Best Practices

- Les cas de tests unitaires doivent être indépendants.
- En cas d'amélioration ou de modification des exigences, les tests élémentaires ne doivent pas être affectés.
- Ne testez qu'un code à la fois.
- Suivez les conventions de dénomination claires et cohérentes pour vos tests unitaires.

# Best Practices

- En cas de changement de code dans un module, assurez-vous qu'il existe un scénario de test d'unité correspondant pour le module et que le module réussisse les tests avant de modifier la mise en œuvre.
- Plus vous écrivez de code sans avoir à tester, plus vous aurez de chemins à vérifier pour détecter les erreurs.

# L'approche TDD (Test-Driven-Development)

Il existe trois approches concernant les tests unitaires :

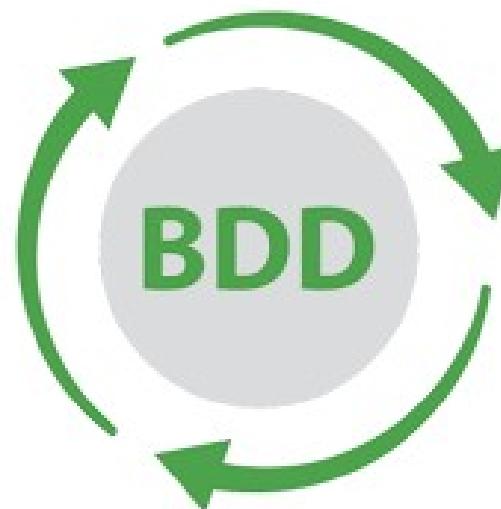
1. Les développeurs n'en écrivent jamais (ne riez pas, c'est malheureusement la majorité des cas) ;
2. Les développeurs écrivent les tests quand ils ont le temps et la motivation (c'est à dire très rarement) ;
3. Les développeurs commencent par écrire les tests avant toute ligne de code métier.

# Les tests unitaires et tests de charge

1. Cette dernière approche porte un nom : le **TDD**, pour Test-Driven-Development. La règle est assez simple : vous commencez par écrire les tests qui permettent de valider les différentes exigences de votre application. Ensuite, vous écrivez le code de votre application qui permet de valider ces tests.
2. Cette méthode permet d'éviter d'écrire trop de code. La base de code est plus saine et moins sujette aux erreurs. D'un autre côté, il n'est pas toujours évident d'avoir cette approche avec des tests d'interface ou lorsqu'une connectivité réseau est présente. Car les paramètres qui rentrent en compte sont multiples.
3. Pour faire simple, cette approche est intéressante pour tester son modèle, et c'est ce que nous allons voir dans les prochains chapitres.



# BDD (Behavior Driven Development)

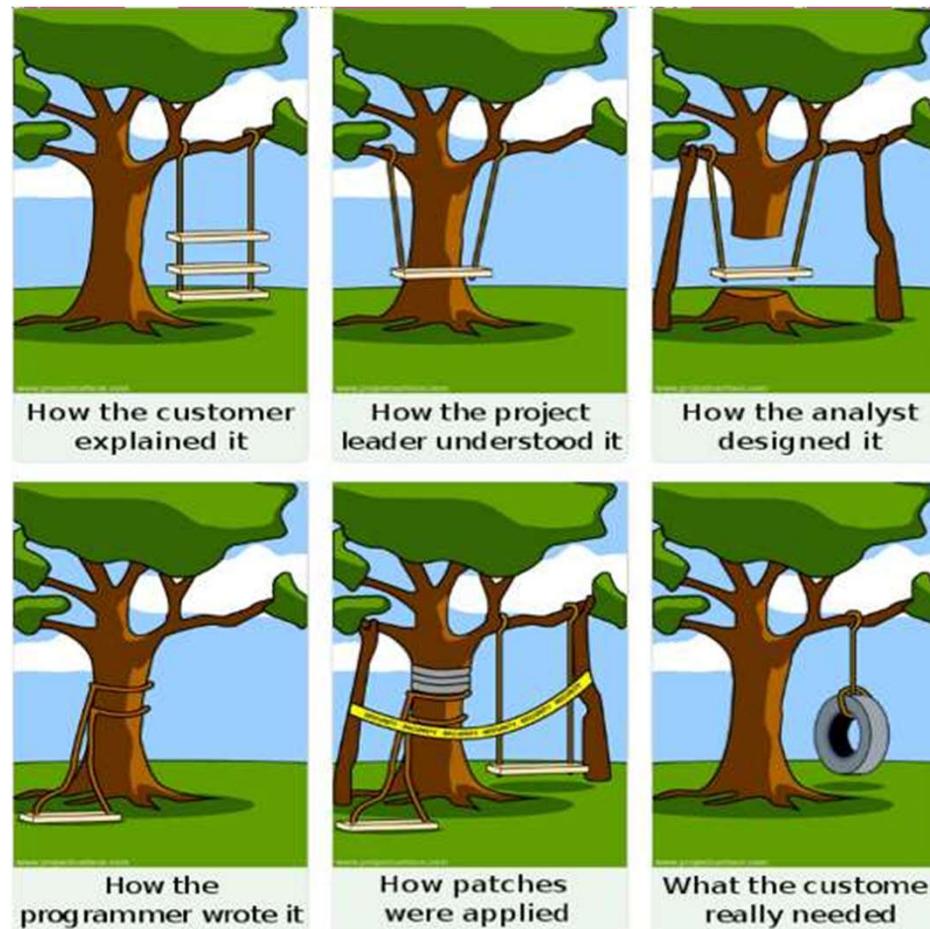


# Ce qu'on entend souvent en entreprise

- Trop de politiques internes (processus lourds)
- Les besoins ne sont pas clairs
- Le client ne sait pas ce qu'il veut
- On fait beaucoup de réunions mais on discute dans le vide
- Le client (ou PO) n'est jamais disponible
- La documentation n'est pas à jour
- La documentation n'est pas claire

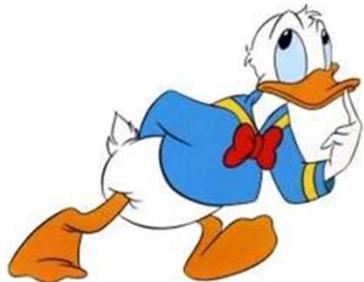


# Ça vous rappelle quelque chose ?



# Et ça ?

Je veux un iPhone



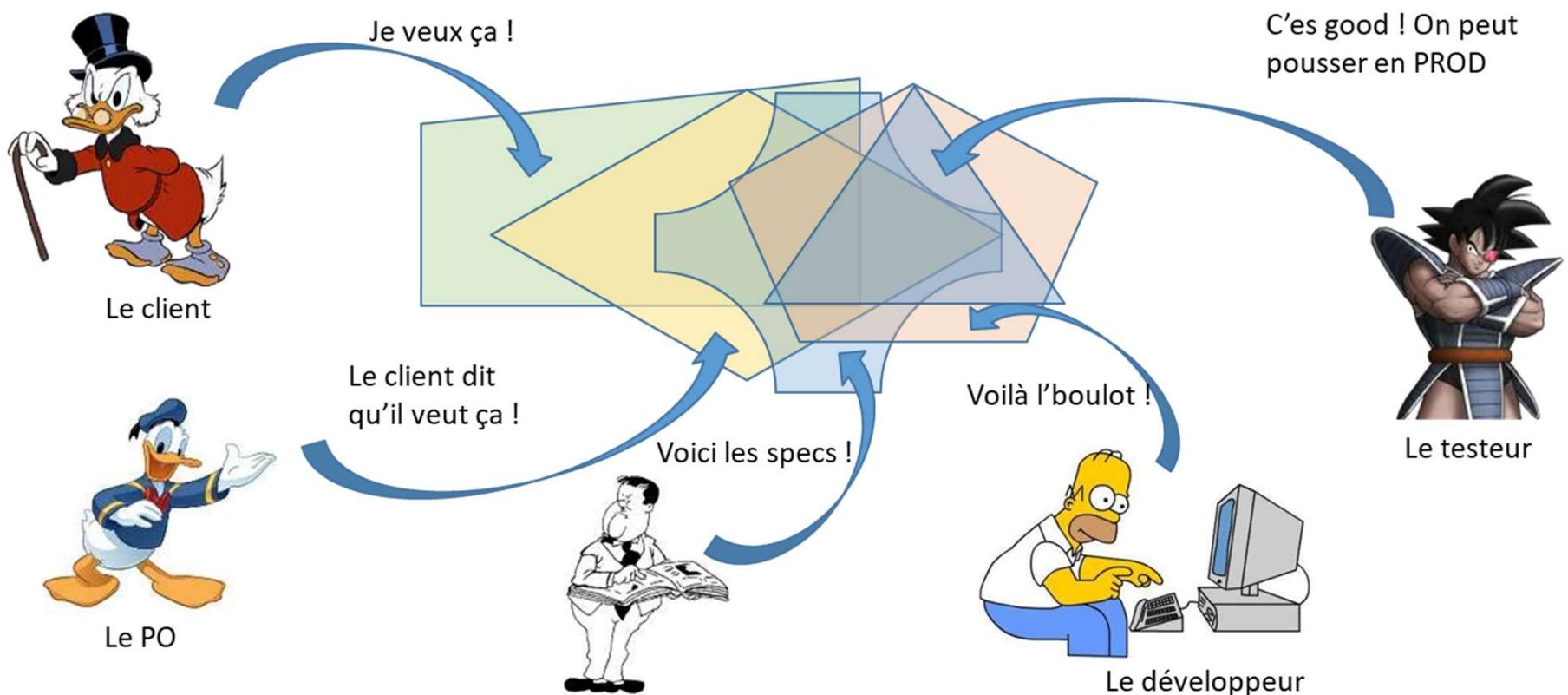
Mais non ! Un téléphone  
Iphone de la marque Apple



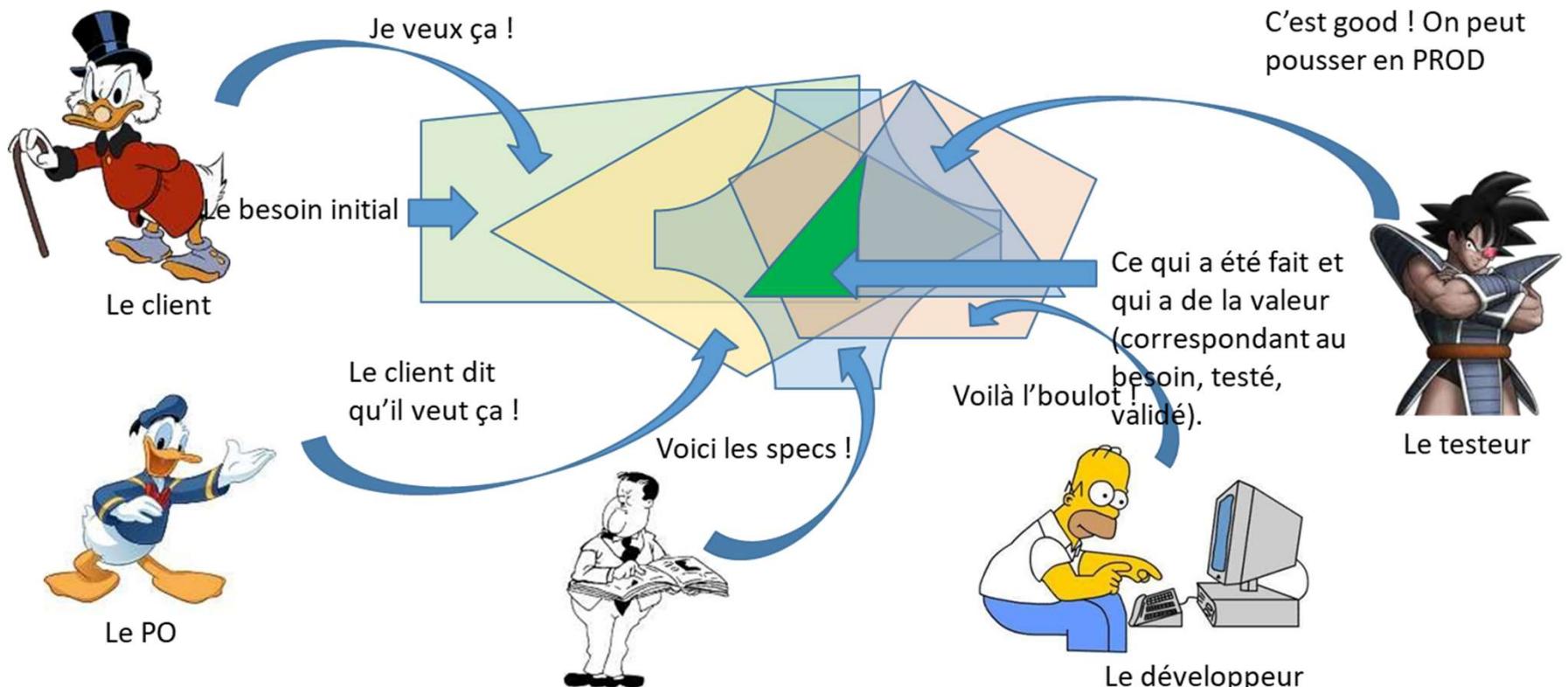
Oui mais non ! Je veux le dernier  
modèle, la version 6



# Ce qu'on voit très souvent



# Ce que ça donne comme résultat



# Pourquoi le BDD ?

- National Institute of Standards and Technology estime à environ 70% le pourcentage d'erreurs introduites par projet lors de la phase d'étude de besoins.
- Réponse au TDD qui est mal compris

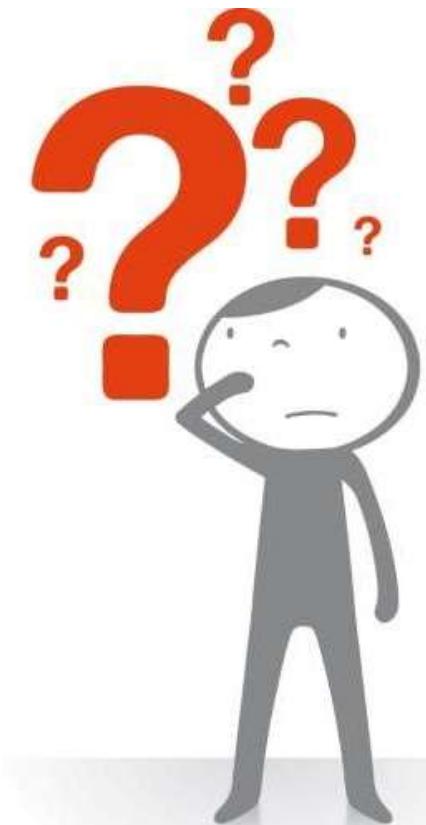
# Ce que certains pensent du BDD

C'est juste une truc à la mode

C'est un truc pour faire des tests (automatisés)

C'est juste du Gerkins (*Given When then*)

Ça ressemble au TDD !



# BDD : objectifs

- Building software right
- Building right software
- Software that matters



# De l'idée à l'implémentation



Je veux **ça** !

Comment on fait **ça** ?  
Comment on fait ça bien ?  
Comment on fait bien **ça** ?

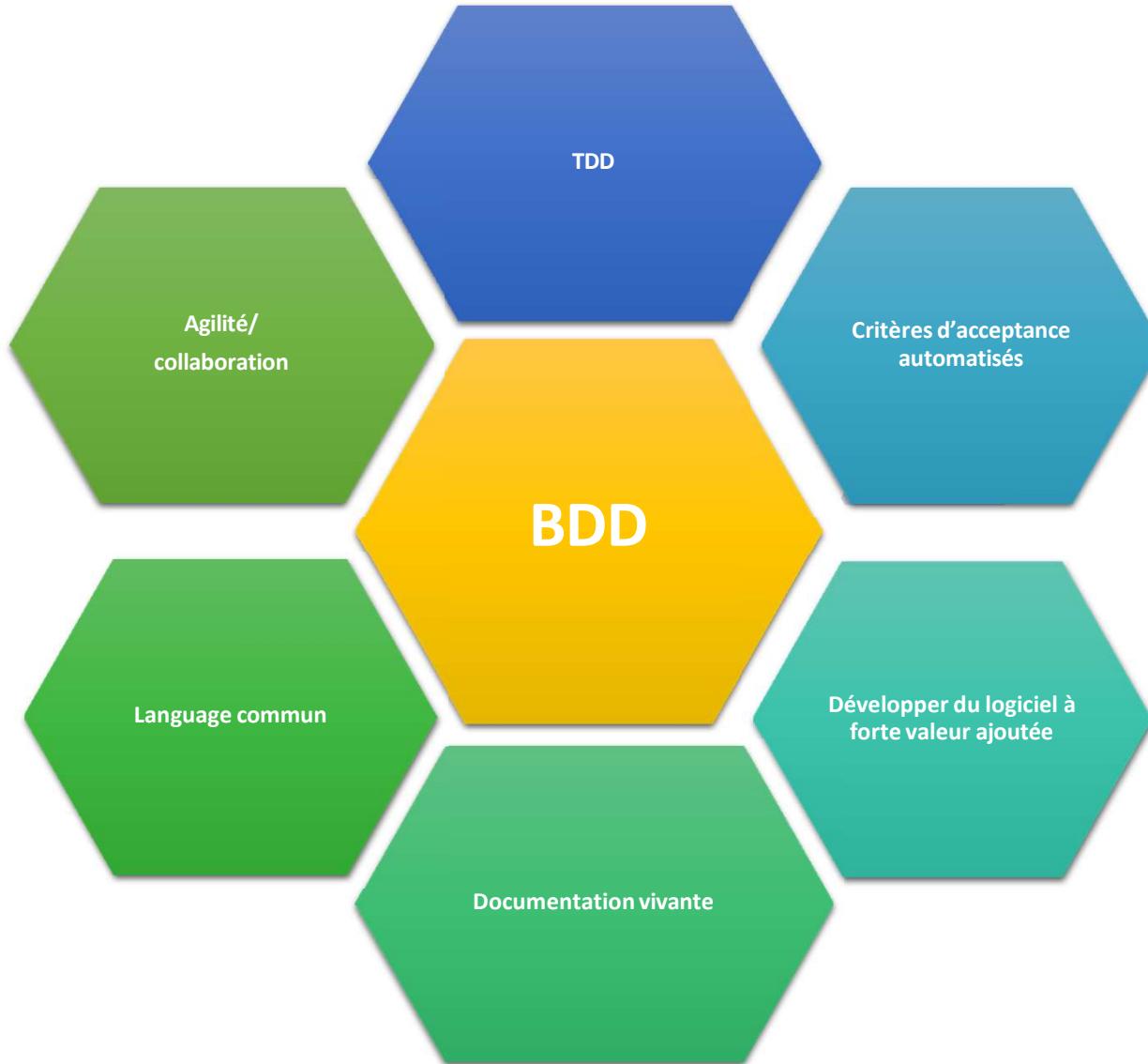


Avec du BDD par exemple



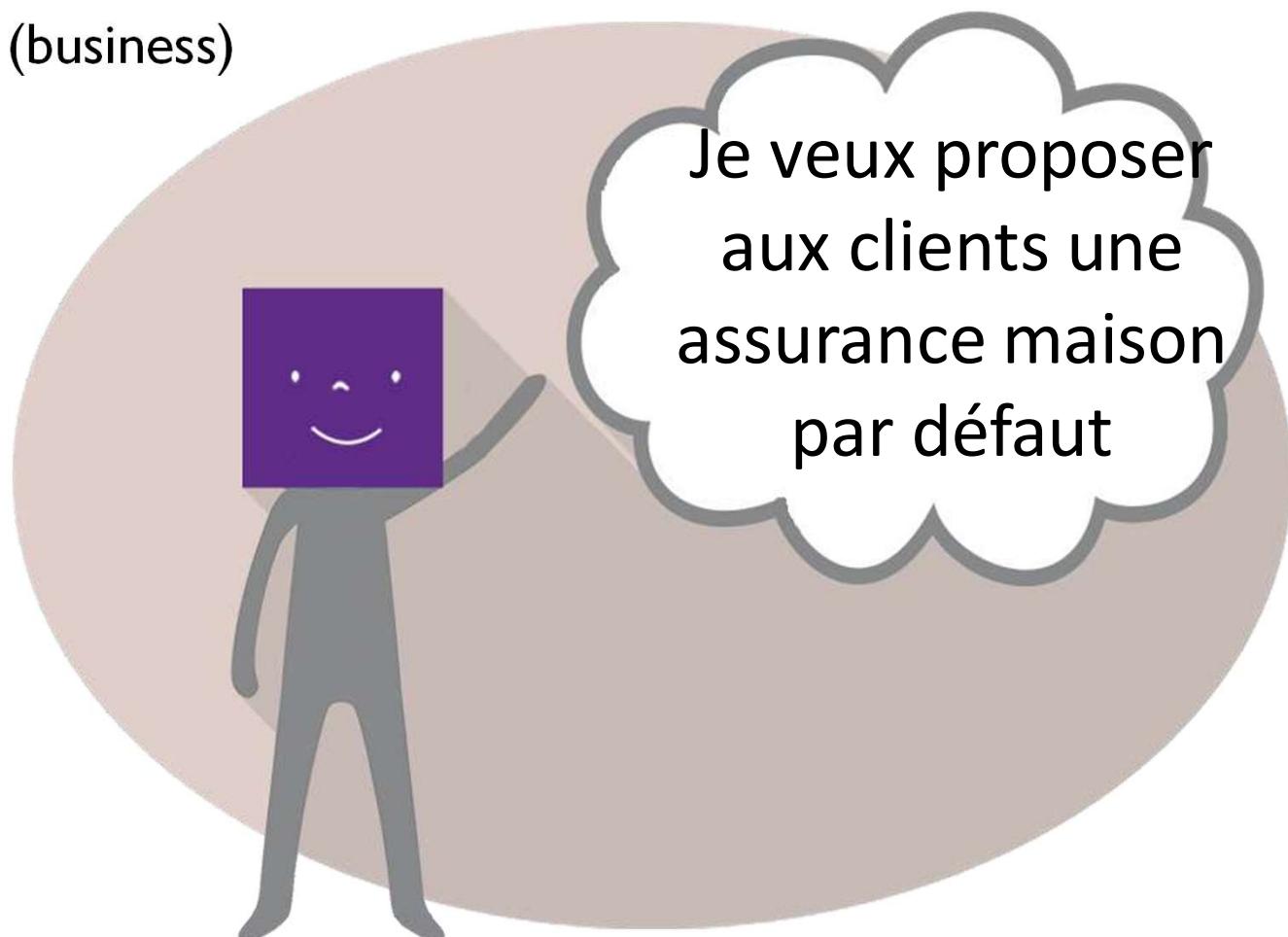
Voilà **ça** !

# Quelques mots...



# Le besoin métier

On commence par définir  
l'objectif métier (business)



# Les features

On répond ensuite à la question :

- Quelles sont les éléments (**features**) qui vont me permettre de répondre à cet objectif ?

Tu veux donc une page web?

Oui.

Tu veux que ton offre d'assurance s'affiche dès la première page ?

Oui, faut que ce soit simple

Ok, mais on affiche quoi ? Le prix de l'assurance ? des images ?

Le prix hors taxe et le prix TTC, d'ailleurs il faudrait que se soit automatiquement appliqué si on change de prix

Ok, il y aura des options que l'utilisateur pourra choisir ?

Oui mais dans un premier temps pas d'option, il faut que je démarre rapidement mon activité.

Ok, ça paraît clair. Il faudra nous donner un exemple avec les tarifs, ça nous aiderait.



# Discussion autour des features

On discute des fonctionnalités à partir d'exemples concrets.

On discute tous ensemble.



# *Ubiquitous language (langage omniprésent)*

- **En tant que** <Utilisateur/vendeur/client,...>
- **Je souhaite** <envoyer un email/consulter mon stock/voir les articles de mon panier,...>
- **Pour** <confirmer ma commande/maîtriser mon stock/passer commande,...>

# Utilisation d'exemples

**Feature:** StringToSimpleName

*Dans le but d'uniformiser les noms de personnes*

*Je voudrais que les noms de famille commencent par des majuscules  
et le reste des lettres en minuscules*

**Scenario Outline:** Formater les noms

**Given** Je saisit un <nom>

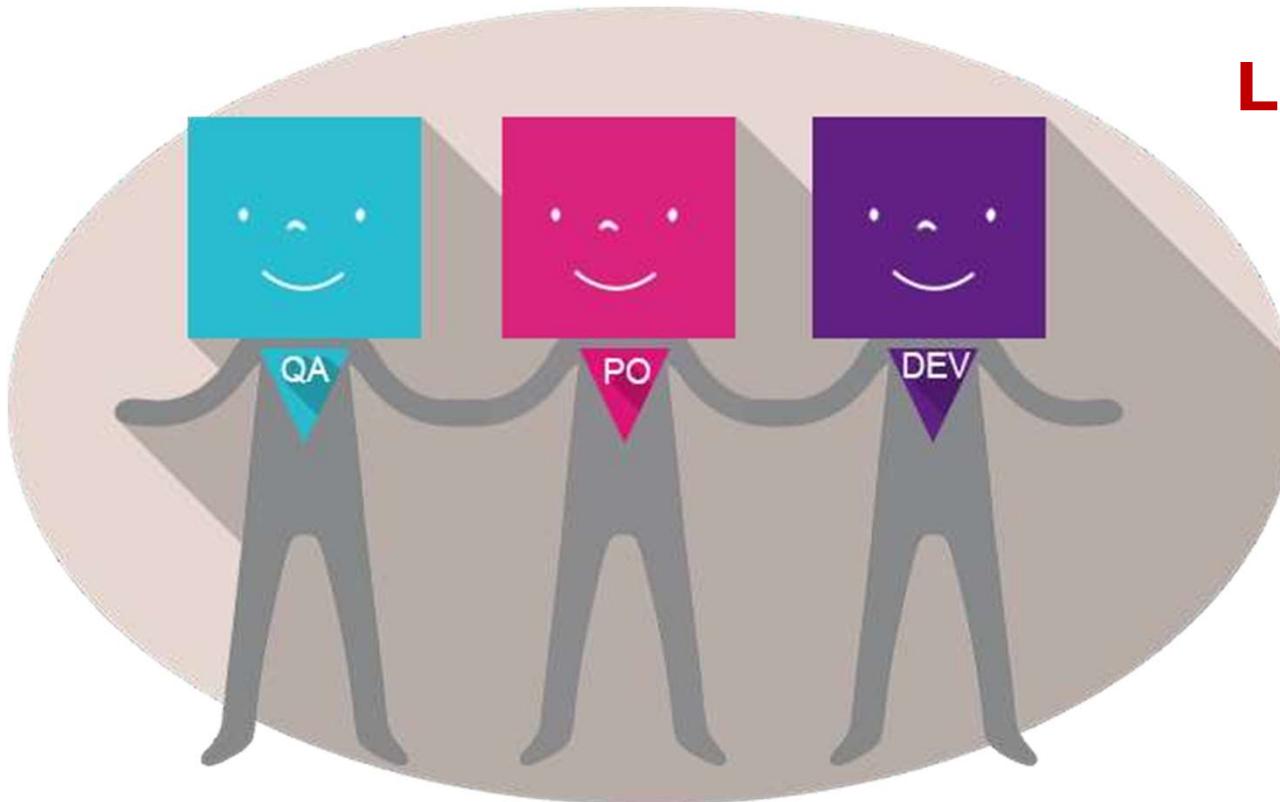
**When** j'appelle la methode de formatage

**Then** Mon <resultat> devrait me retourner un nom commençant par une majuscule

**Examples:**

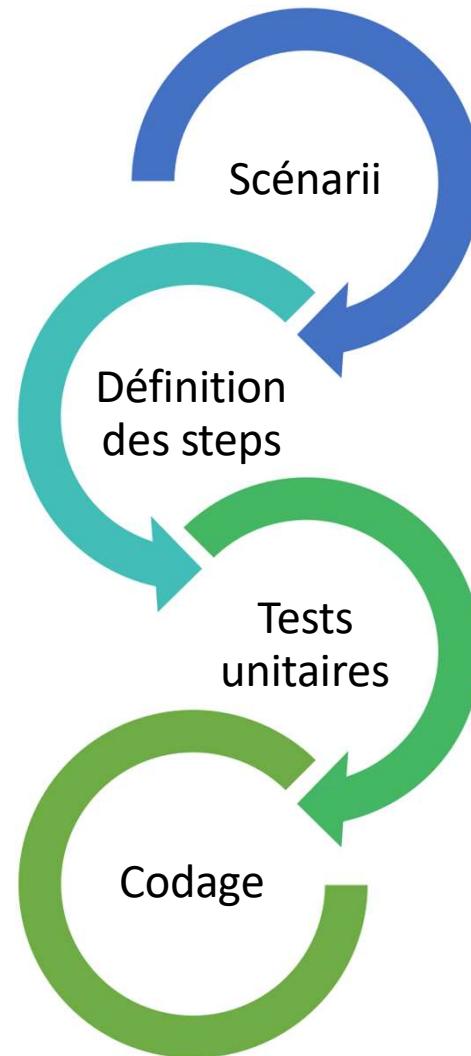
<i>nom</i>	<i>resultat</i>
duPont	Dupont
Dupuis	Dupuis

# Compréhension partagée



**Les 3 amis**

# BDD : les phases



# BDD : Feature

**Feature:** DisplayInsurances

*In order to subscribe insurance  
As a prospect  
I want to see home insurance*

# BDD : scénarii

Scenario: Show insurance

When I open insurance page

Then the result should show the home  
insurance

And the base price is 200

And the sales price is 240

# BDD : Step definition

```
[Binding]
public class DisplayInsurancesSteps
{
    [When(@"I open insurance page")]
    public void WhenIOpenInsurancePage()
    {}

    [Then(@"the result should show the home insurance")]
    public void ThenTheResultShouldShowTheHomeInsurance()
    {}

    [Then(@"the base price is (.*)")]
    public void ThenTheBasePriceIs(float basePriceExpected)
    {}

    [Then(@"the sales price is (.*)")]
    public void ThenTheSalesPriceIs(float salesPriceExpected)
    {}
}
```

# BDD : unit test (TDD)

```
[When(@"I open insurance page")]
public void WhenIOpenInsurancePage()
{
    var homeController = new HomeController();
    this.actionResult = homeController.Index() as ViewResult;
}

[Then(@"the result should show the home insurance")]
public void ThenTheResultShouldShowTheHomeInsurance()
{
    var viewResult = this.actionResult as ViewResult;
    Assert.IsInstanceOf<InsuranceViewModel>(viewResult.Model);
    this.insurance = viewResult.Model as InsuranceViewModel;
}
```

# BDD : unit test (TDD)

```
[Then(@"the base price is (.*)")]
public void ThenTheBasePriceIs(float basePriceExpected)
{
    Assert.AreEqual(basePriceExpected, this.insurance.BasePrice);
}

[Then(@"the sales price is (.*)")]
public void ThenTheSalesPriceIs(float salesPriceExpected)
{
    Assert.AreEqual(salesPriceExpected, this.insurance.SalesPrice);
}
```

# BDD : codage

The screenshot shows the Visual Studio interface with the Test Explorer window open. The Test Explorer shows three passed tests: CheckFireOp..., ShowFireOption, and ShowInsurance. The main code editor area displays a BDD feature file named DisplayInsurancesSteps.cs. The feature is titled "Feature: DisplayInsurances" and describes a scenario where a prospect wants to see home insurance. Three scenarios are defined: "Scenario: Show insurance", "Scenario: Show Fire Option", and "Scenario: Check Fire Option". Each scenario lists its steps, such as opening the insurance page, showing the result, and checking price values.

```
Test Explorer InsuranceDreamManager.cs Insurance.cs DisplayInsurancesSteps.cs

Feature: DisplayInsurances
  In order to subscribe insurance
  As a prospect
  I want to see home insurance

  Scenario: Show insurance
    When I open insurance page
    Then the result should show the home insurance
    And the base price is 200
    And the sales price is 240

  Scenario: Show Fire Option
    When I open insurance page
    Then the result should show the home insurance
    And the Fire Option is display with empty checkbox
    And the Fire Option is 30

  Scenario: Check Fire Option
    When I check fire option
    Then the result should refresh the home insurance
    And the base price is 230
    And the sales price is 276
```

# BDD & agilité

Les 4 valeurs de l'agilité :

- les individus et leurs interactions plus que les processus et les outils,
- du logiciel qui fonctionne plus qu'une documentation exhaustive,
- la collaboration avec les clients plus que la négociation contractuelle
- l'adaptation au changement plus que le suivi d'un plan.

# Des avantages : le bon résultat



ON DÉVELOPPE LE BON  
PRODUIT

ON LE DÉVELOPPE DE LA  
MEILLEUR FAÇON

# Des avantages : des tests automatisés



Les critères  
d'acceptances  
sont automatisés

Les tests sont  
automatisés

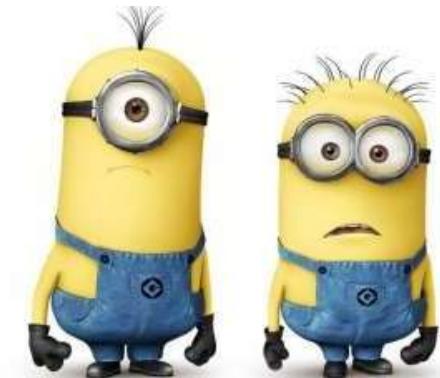
# Des avantages : une documentation vivante et à jour

- La documentation est à jour
- Les scenarii/features reflètent le besoin réel du client
- Le code correspond aux tests qui dérivent des features implémentées



# Des difficultés et un challenge

- Faire participer tout le monde
- Montrer « aux chefs » que ça marche
- Faire adopter la pratique par les participants
- Faire accepter de changer de méthode de travail
- Faire preuve d'empathie
- Pratique testée mais pas adoptée



# Les tests unitaires et tests de charge

