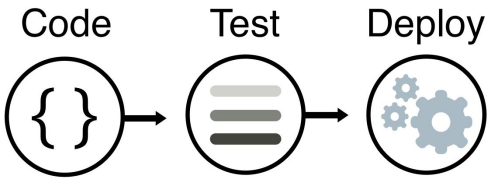
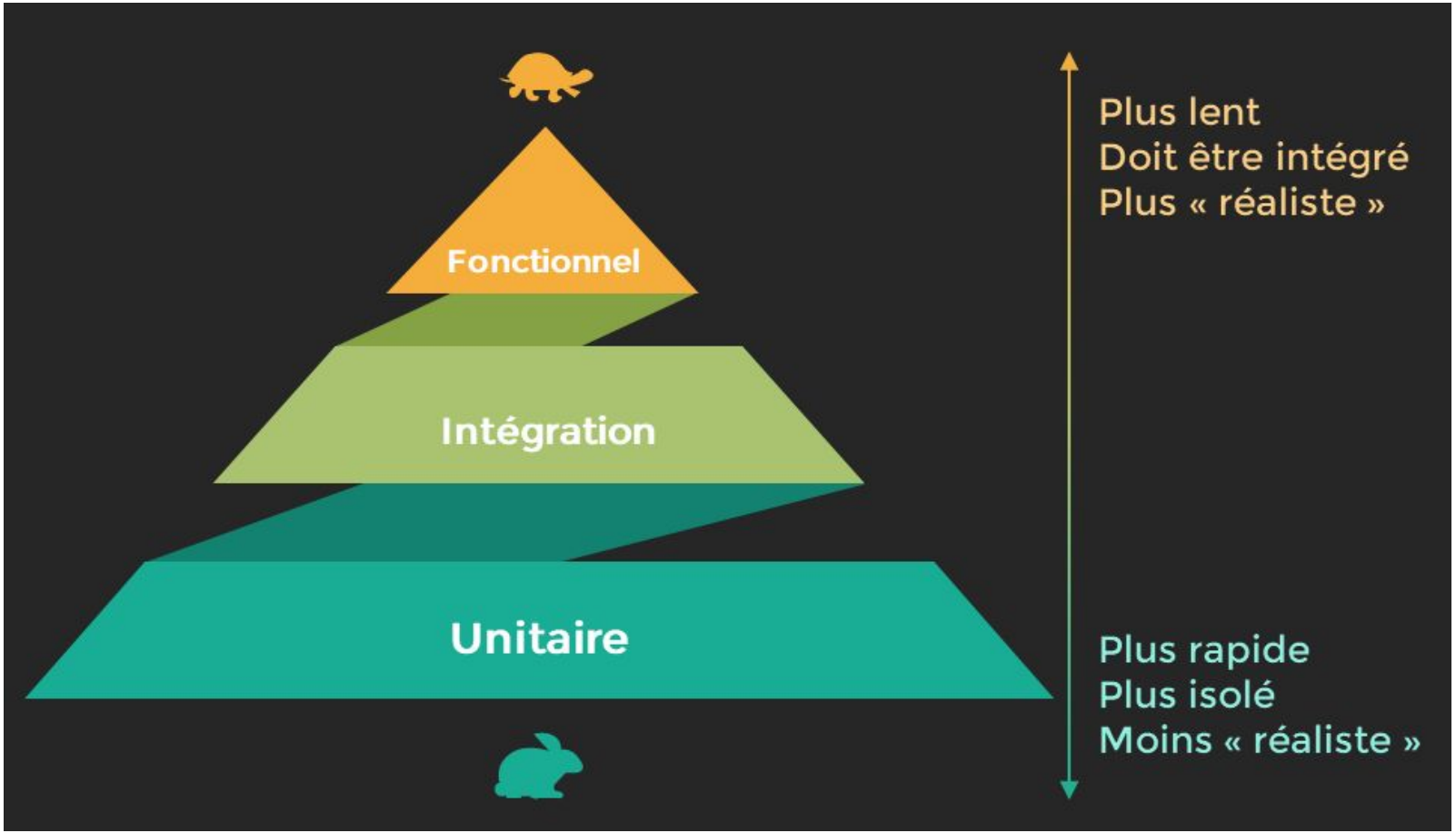
Code Testing

Using java

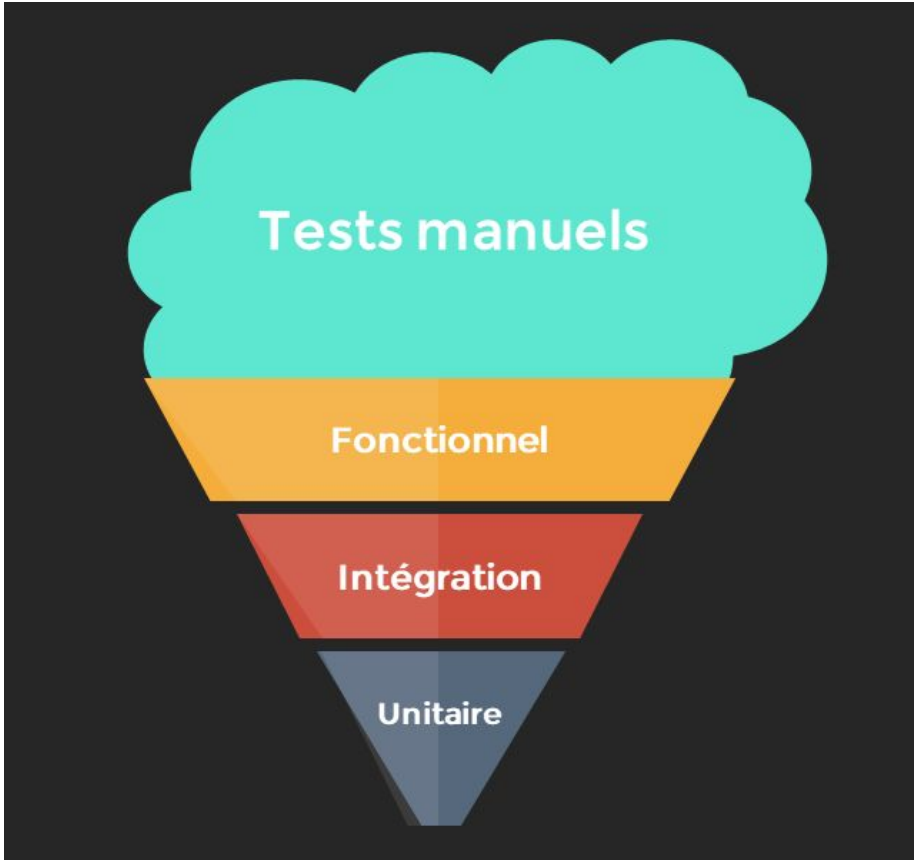


Marwén Saidi - marwen.saidi@orange.com

En quoi consiste les tests : La pyramide des tests



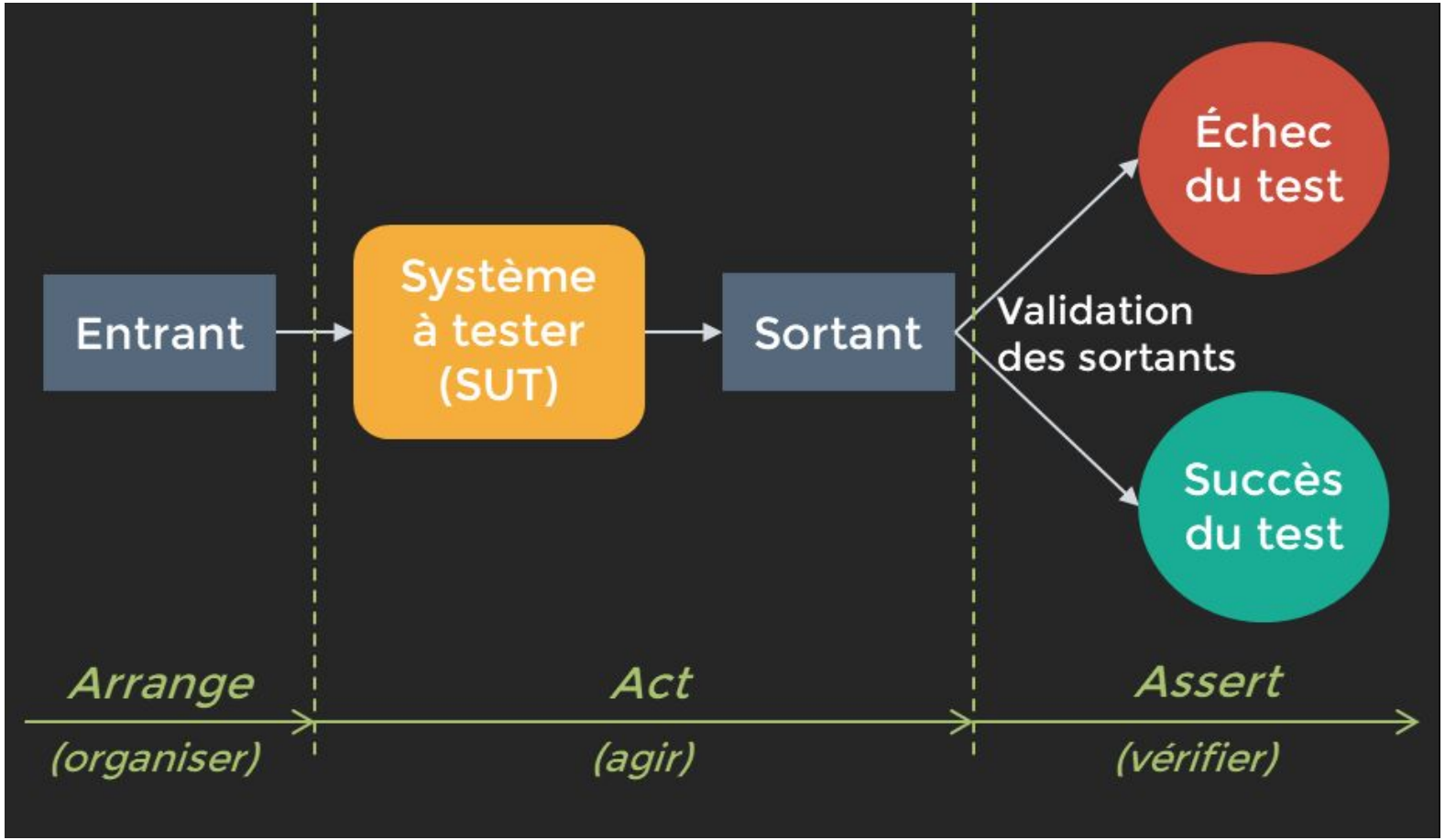
Les mauvaises pratiques dans la réalité des projets : Anti-pattern du cône de glace de tests



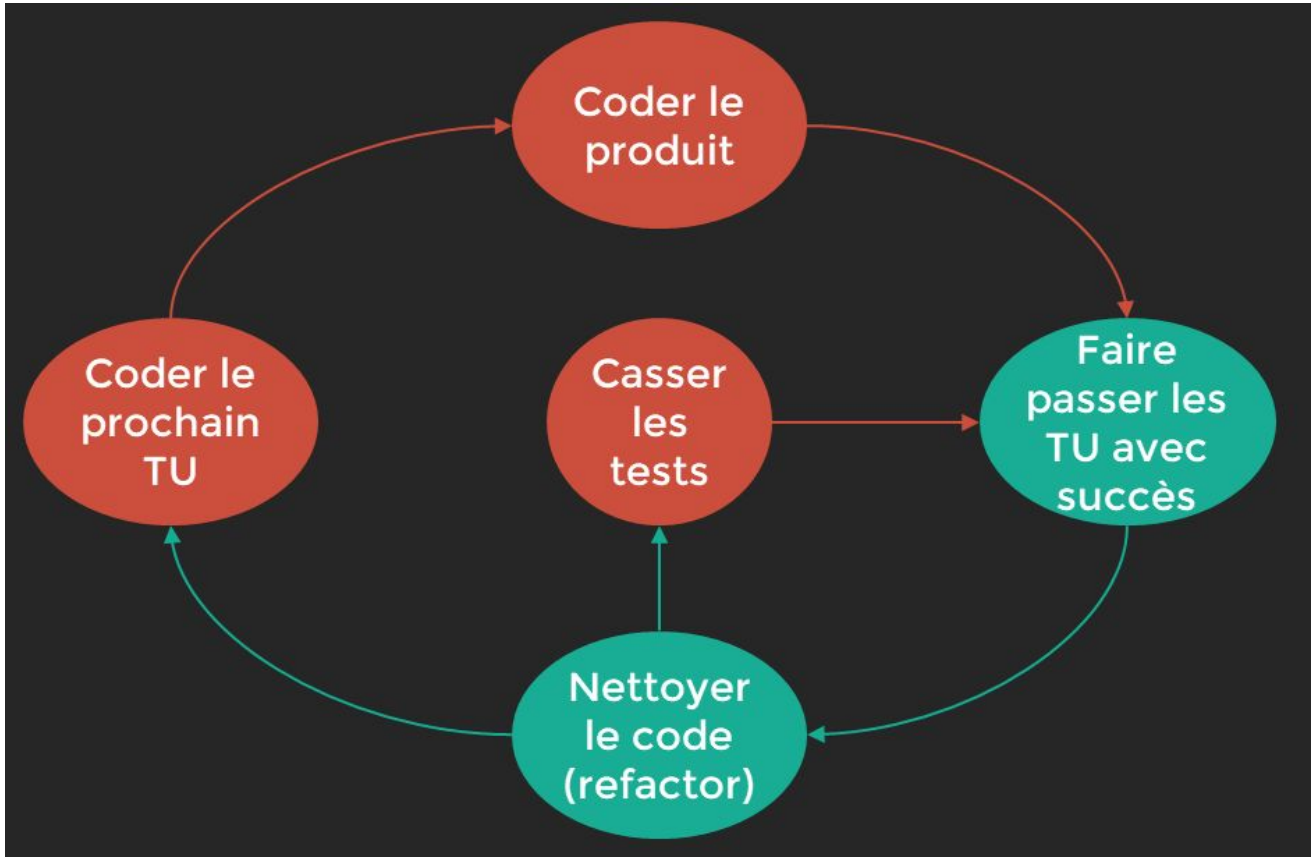
Découvrez d'autres finalités pour les tests

- Testez pour faire face à l'inattendu
- Testez pour faciliter la maintenance
- Testez pour communiquer

SUT : system under test



Utilisez le TDD : rouge-vert-refactor !

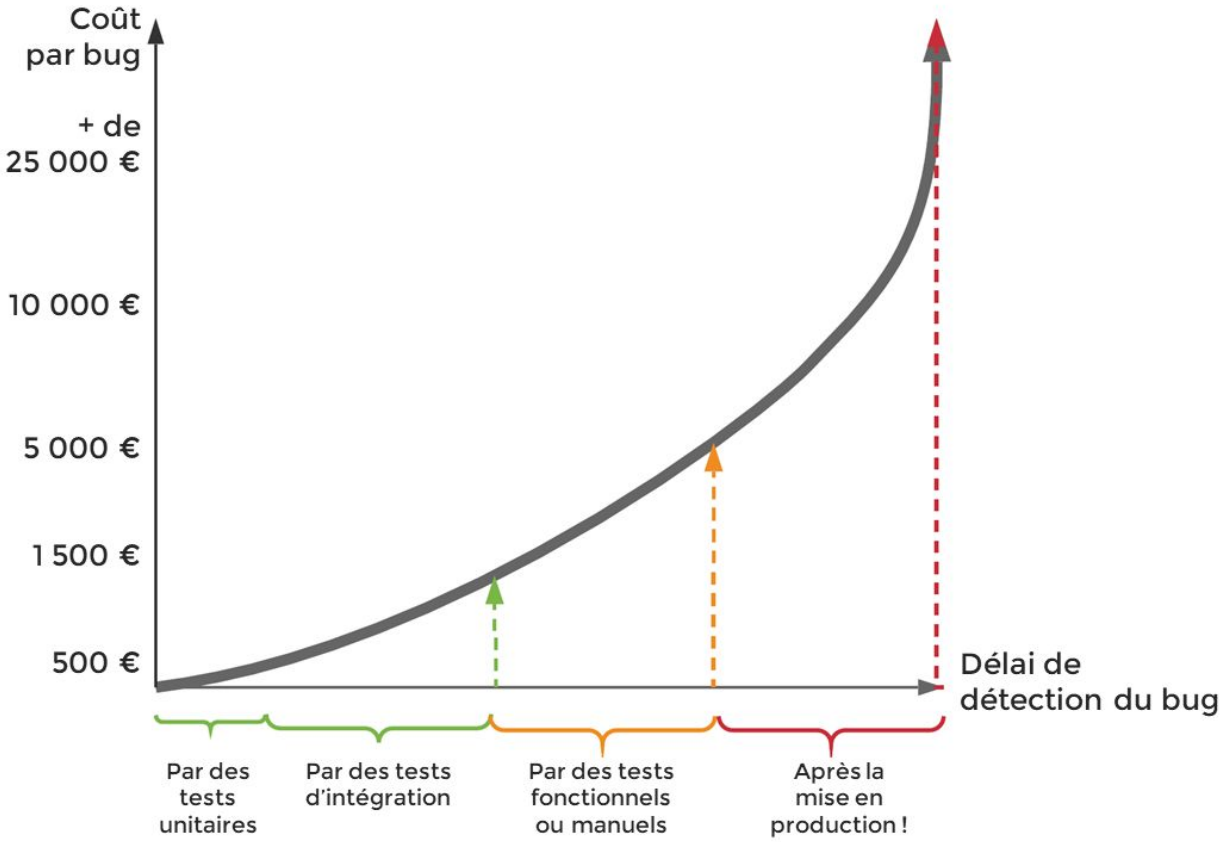



Demo
Time!

Testing Annotation

Annotation	Quand l'utiliser
@BeforeEach	Exécutez une méthode avant **chaque** test. C'est un très bon emplacement pour installer ou **organiser** un prérequis pour vos tests.
@AfterEach	Exécutez une méthode après **chaque** test. C'est un très bon emplacement pour nettoyer ou satisfaire à une postcondition.
@BeforeAll	Désignez une méthode statique pour qu'elle soit exécutée avant *tous* vos tests. Vous pouvez l'utiliser pour installer d'autres variables statiques pour vos tests.
@AfterAll	Désignez une méthode statique pour qu'elle soit exécutée après *tous* vos tests. Vous pouvez utiliser ceci pour nettoyer les dépendances statiques.
@ParametrizedTest	Vous souhaitez réutiliser le même test avec plusieurs entrants (**@ValueSource**) voire plusieurs entrants/sortants (**@CsvSource**).
@Timeout	Si vous testez une méthode qui ne doit pas être trop lente, vous pouvez la forcer à échouer le test.

Coût par bug à mesure que le temps passe

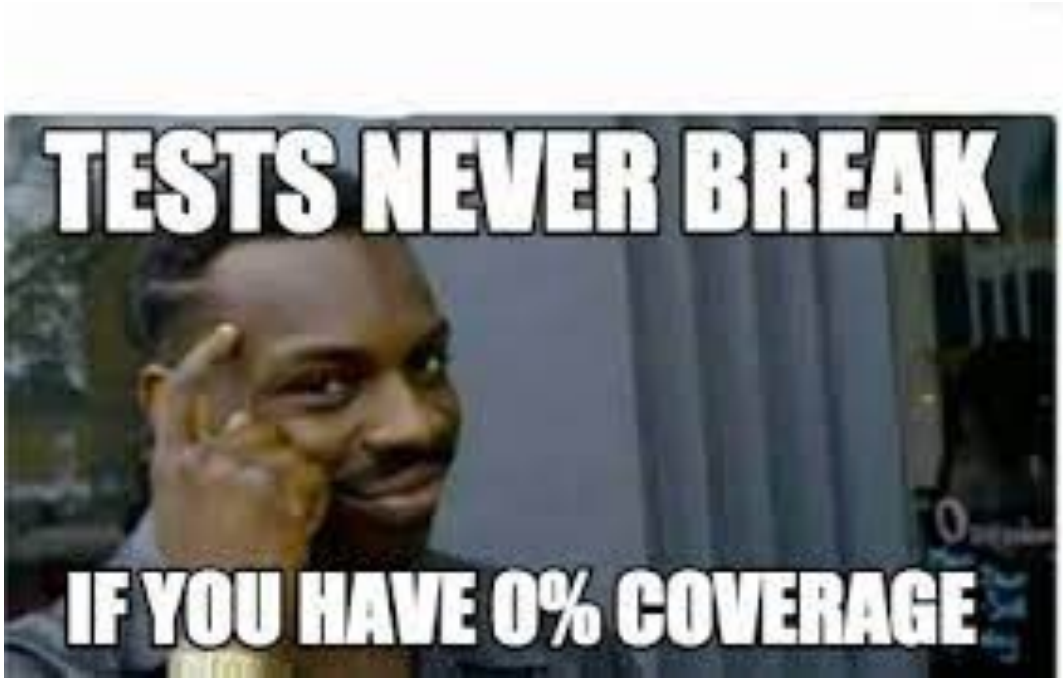


AssertJ

Cas de test	Assertions JUnit	Assertions AssertJ
Un nom est compris entre 5 et 10 caractères.	assertTrue(name.length > 4 && name.length < 11);	assertThat(name) .hasSizeGreaterThan(4) .hasSizeLessThan(11);
Un nom est situé dans la première moitié de l'alphabet	assertTrue(name.compareTo("A") >= 0 && name.compareTo("M") <= 0);	assertThat(name).isBetween("A", "M");
Une date et heure locale se situent aujourd'hui ou dans le futur.	assertTrue(dateTime.toLocalDate().isAfter(LocalDate.now()) ‖ dateTime.toLocalDate().isEqual(LocalDate.now()));	assertThat(dateTime.toLocalDate()) .isAfterOrEqualTo(LocalDate.now()););

Test Coverage

- IDE test coverage
- Maven automatisation
- SonarLint
- SonarQube



Tests tagged with annotation

~~@Category~~ maintenant on utilise @Tag

- @Tag désigne tous les tests de la classe comme étant des tests de conversion, avec un tag nommé "ConversionTests".
- @DisplayName vous permet de nommer vos tests de façon lisible par tous.
- @Nested vous permet de grouper vos tests dans une classe interne. Avec @Nested, si un seul test échoue, tout le groupe désigné par cette annotation échoue !
- vous pouvez ajouter @Displayname et @Tag à chaque bloc @Test et @Nested.

Other tag

- **@ExtendWith** peut être utilisé pour modifier le déroulement des tests.
- **@Disabled** peut être utilisé pour faire taire **temporairement** les tests inutiles ou problématiques.

F.I.R.S.T

PRINCIPE	Qu'est-ce que cela signifie ?
Fast (Rapide)	Une demi-seconde, c'est déjà trop lent. Vous avez besoin que des milliers de tests soient exécutés en moins de quelques secondes.
Isolé et **i**ndépendant	Quand un test échoue, il vous faut savoir ce qui a échoué. Testez un élément à la fois et n'utilisez plus d'une assertion que s'il le faut vraiment.
Répétable	Vous ne pouvez pas vous fier aux tests qui échouent certaines fois et réussissent d'autres. Vous devez construire des tests qui n'interfèrent pas les uns avec les autres et qui sont assez autonomes pour donner toujours le même résultat.
Self-validating (Autovalidation)	Utilisez les bibliothèques d'assertions disponibles, écrivez des tests spécifiques, et laissez votre framework de test s'occuper du reste. Vous pouvez vous y fier pour exécuter vos tests et générer des rapports de tests clairs.
Thourough (Approfondi)	Utilisez le TDD et écrivez vos tests en écrivant votre code. Explorez toute la gamme d'actions possibles de votre méthode et écrivez beaucoup de tests unitaires.

Scénario inattendu … comment tester ?

Comment trouver de bons cas limites à tester ?

Voici de bonnes questions à poser lorsque l'on recherche des cas limites :

- Quelle serait une valeur saugrenue à passer dans cette méthode, qui serait néanmoins légalement permise par son type défini ? Par exemple, passer une chaîne nulle, vide, ou incroyablement longue à Integer.parseInt(String) ?
- Mon application fonctionnera-t-elle encore dans quelques années ? Est-ce que mes règles business autorisent des volumes de data croissants (par exemple d'en collecter davantage) ? Est-ce que j'ai des compteurs ou des chaînes qui vont évoluer jusqu'à poser problème par la suite ?
- Si vous écrivez une méthode, demandez-vous : « Que pourrais-je essayer de faire pour la casser ? » Cela nécessite-t-il des arguments que je pourrais essayer de passer avec des valeurs inhabituelles ? Comme des valeurs nulles, des objets mal configurés, ou des chaînes vides, par exemple.
- Dois-je coder avec un degré de précision défini pour mes cas de test (par exemple, exactitude à deux décimales près) ? Comment doit se comporter mon code s'il doit arrondir plusieurs fois ? Par exemple, diviser par deux et arrondir ; @Test diviser par deux et arrondir à nouveau. Serait-il toujours assez précis ?

- Les tests de **cas limites** (« **edge cases** ») sont des tests conçus pour vérifier l'inattendu aux limites de votre système et de vos limites de données.
- Les tests de **cas pathologiques** (« **corner cases** ») testent des situations improbables dans lesquelles votre application pourrait se retrouver.
- Prenez en compte vos **règles métiers** et **limites techniques** pour vous guider dans la définition des cas de tests improbables.

Mockito

Pour mocker efficacement les objets autour de votre système à tester, voici la démarche à suivre :

1. Identifiez un comportement unique que vous testez avec votre classe sous-test (CUT).
2. Demandez-vous quelles classes sont nécessaires au comportement à tester.
3. Hormis votre CUT, envisagez toutes les autres classes pour le mocking.
4. Ne mockez pas les classes qui ne servent quasiment qu'à porter des valeurs.
5. Installez les mocks requis.
6. Testez votre CUT.
7. Vérifiez que vos mocks ont été correctement utilisés.

Mockito

- Dans les tests de logiciel, on utilise habituellement un acteur de remplacement pour jouer le rôle de la classe que vous avez besoin d'utiliser ; ce remplaçant s'appelle une ***doublure de test*** ou *test double* en anglais.
- Un **mock**, ou une **simulation**, est un type de doublure de test simple à créer, et qui vous permet également de tester comment on interagit avec lui.
- **Mockito** vous permet de définir **comment vos mocks doivent se comporter** (avec when) et **vérifier si ces mocks sont bien utilisés** (avec verify).
- En faisant du TDD, vous pouvez être amené à coder des évolutions avec des services complémentaires. Mockito vous aide à pratiquer le TDD facilement grâce aux mocks.

Mockito … advanced

Situation	**Mock**	**Espion**
Créer avec l'annotation	@Mock	@Spy
Créer avec une méthode	**mock(RealClass.class)**	**spy(RealClass.class)**
Appeler **someMethod** après **when(someMethod)** **.thenReturn(response)**	Renvoie la **réponse** fournie	Renvoie la **réponse** fournie
Appeler **someMethod()** que vous n'avez **pas** définie avec 'when'	Le mock renvoie un **null**	La méthode originale est appelée dans la **RealClass** pour un résultat
Capturer des arguments avec **ArgumentCaptor**	**verify(mock)** **.someMethod(captor)**	**verify(spy)** **.someMethod(captor)**
Simuler une méthode ou classe **finale**	Erreur de Mockito	Erreur de Mockito

Mockito … advanced

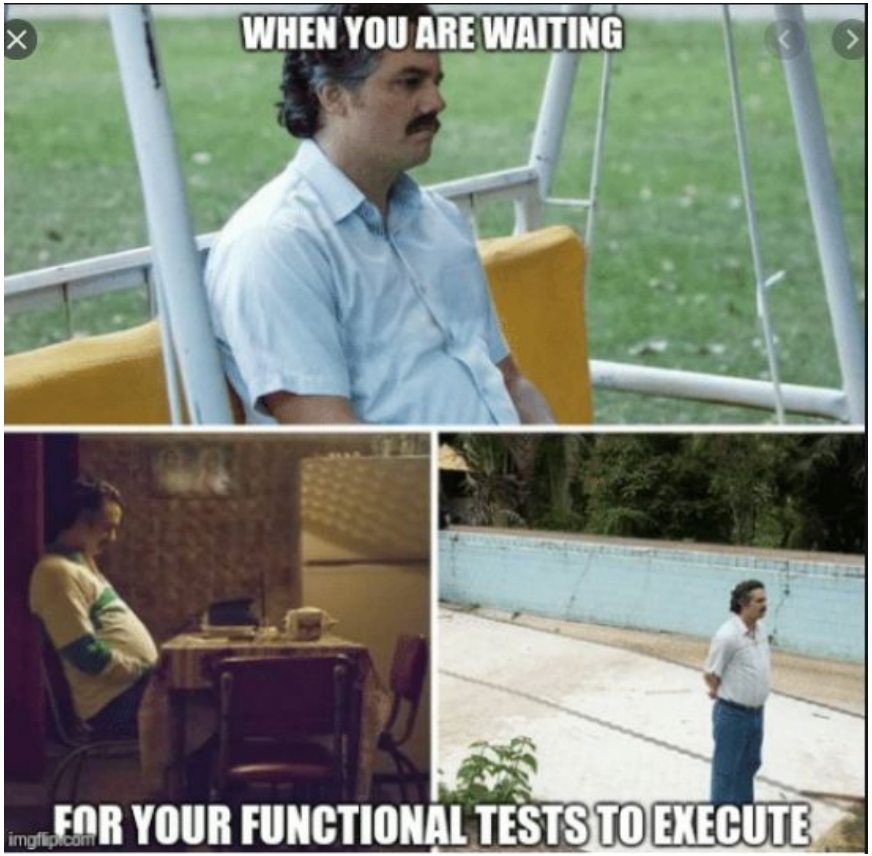
- Utilisez **ArgumentCaptor** pour capturer les arguments réels avec lesquels votre mock est appelé. Cela vous permet de vérifier la conformité des appels aux mocks. Et ce, en particulier lorsque votre classe de test utilise des arguments sur lesquels vous n'avez pas de visibilité directe.
- Utilisez les fonctions de réponse sous forme de lambdas ou les réponses multiples avec when(), pour obtenir des comportements plus sophistiqués de vos mocks.
- **@Spy** et **spy()** vous permettent d'utiliser des instances réelles d'une classe, mais sur lesquelles vous pouvez utiliser when() et verify(). Ce sont des mocks partiels. Cela se révèle nécessaire lorsque vos classes contiennent des méthodes avec le modificateur final.

Integration Tests

- **les tests d'intégration composants** : ils permettent de vérifier si plusieurs unités de code fonctionnent bien ensemble, dans un environnement de test assez proche du test unitaire, c'est-à-dire de manière isolée, sans lien avec des composants extérieurs et ne permettant pas le démarrage d'une vraie application ;
- **les tests d'intégration système** : ils permettent de vérifier le fonctionnement de plusieurs unités de code au sein d'une configuration d'application, avec éventuellement des liens avec des composants extérieurs comme une base de données, des fichiers, ou des API en réseau.

Functional Tests

tests fonctionnels de bout en bout. Ce sont des tests qui partent de l'interface utilisateur pour obtenir un résultat selon un scénario prédéfini. Ils imitent l'utilisateur final de l'application. Un démarrage complet de l'application est donc nécessaire.



Acceptance Tests

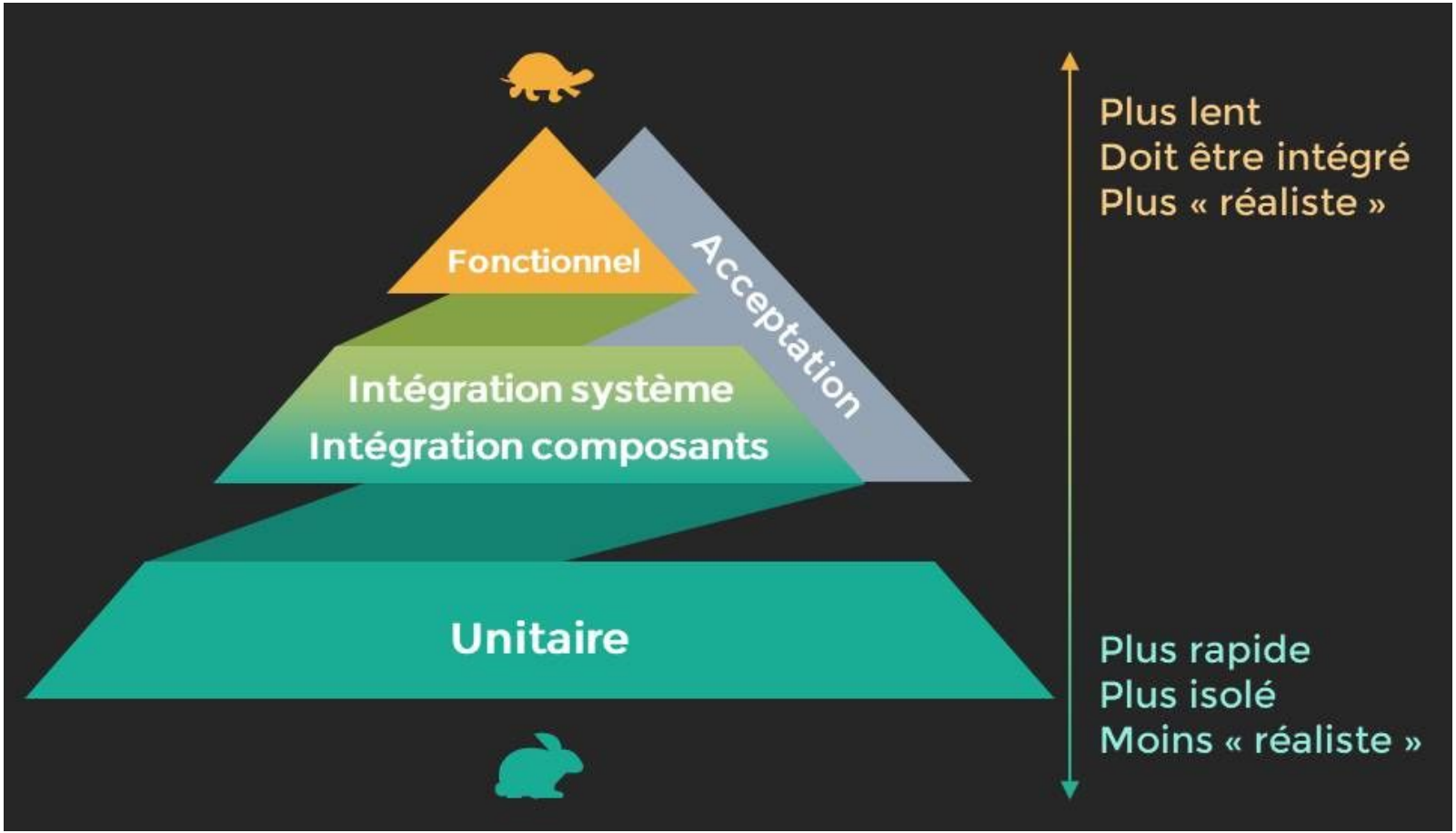
En complément des tests d'intégration et des tests fonctionnels, vous pouvez effectuer des tests d'acceptation. Ils abordent un autre point de vue : il s'agit de vérifier directement une exigence métier, exprimée par le client, un expert fonctionnel le représentant.

Pyramide	Test	Question à laquelle on répond	Exemple
Au milieu ou au sommet selon le cas	Tests d'acceptation	Comment vérifier une exigence métier de manière automatisée ?	Un test qui décrit les exigences du métier, et non les composants. L'implémentation peut varier.

Other tests type

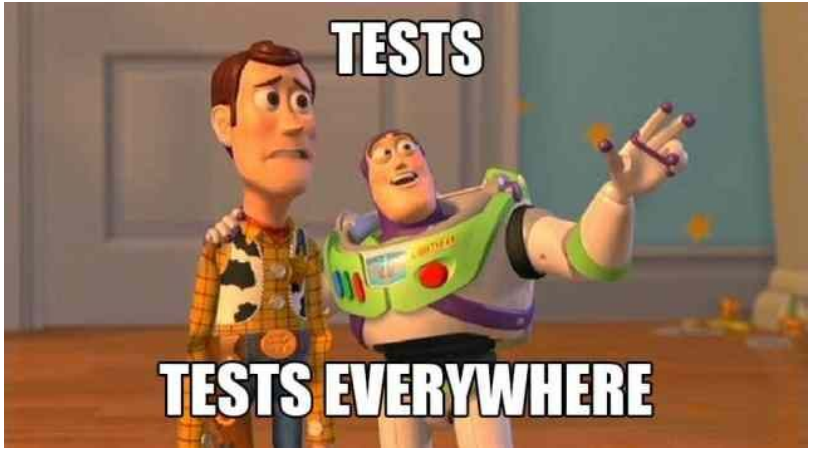
Test	Question à laquelle on répond	Exemple
Bêta	Que se passe-t-il quand je présente une version de mon application à des utilisateurs réels ?	Si je publie mon calculateur pour des élèves réels de l'école locale, est-ce qu'il est performant et utilisable ?
A/B testing	J'ai deux idées géniales pour cette interface utilisateur ou un bout de code compliqué. Laquelle devrais-je utiliser ?	Vous pourriez sortir deux versions de votre application. L'une enverrait tous les calculs à Google, et l'autre les résoudrait elle-même. Comparez laquelle est meilleure et préférée par vos utilisateurs.
Smoke testing	L'intégration entre votre application en marche (tout le code) et la plateforme sur laquelle elle fonctionne.	L'application fonctionne sur votre plateforme de production. Les tests rapides vérifient que ses configurations sont correctes et qu'elle semble gérer un ou deux bons chemins.

to summarize



to summarize more

- Pour que les **tests unitaires** soient rapides et approfondis, ils formulent de **nombreuses hypothèses,** exprimées à travers les mocks.
- Les **tests d'intégration des composants** valident qu'un petit nombre d'unités de code peuvent collaborer ensemble.
- Les **tests d'intégration système** valident la capacité de vos unités de code à collaborer dans un système partiellement en fonction, avec des composants extérieurs simulés ou non.
- Avec les **tests de bout en bout**, vous faites fonctionner le système complet et vous validez vos hypothèses.
- Les **tests d'acceptation** fonctionnent de façon perpendiculaire à la pyramide et peuvent traverser les niveaux, tout en restant en principe au milieu ou en haut de la pyramide.

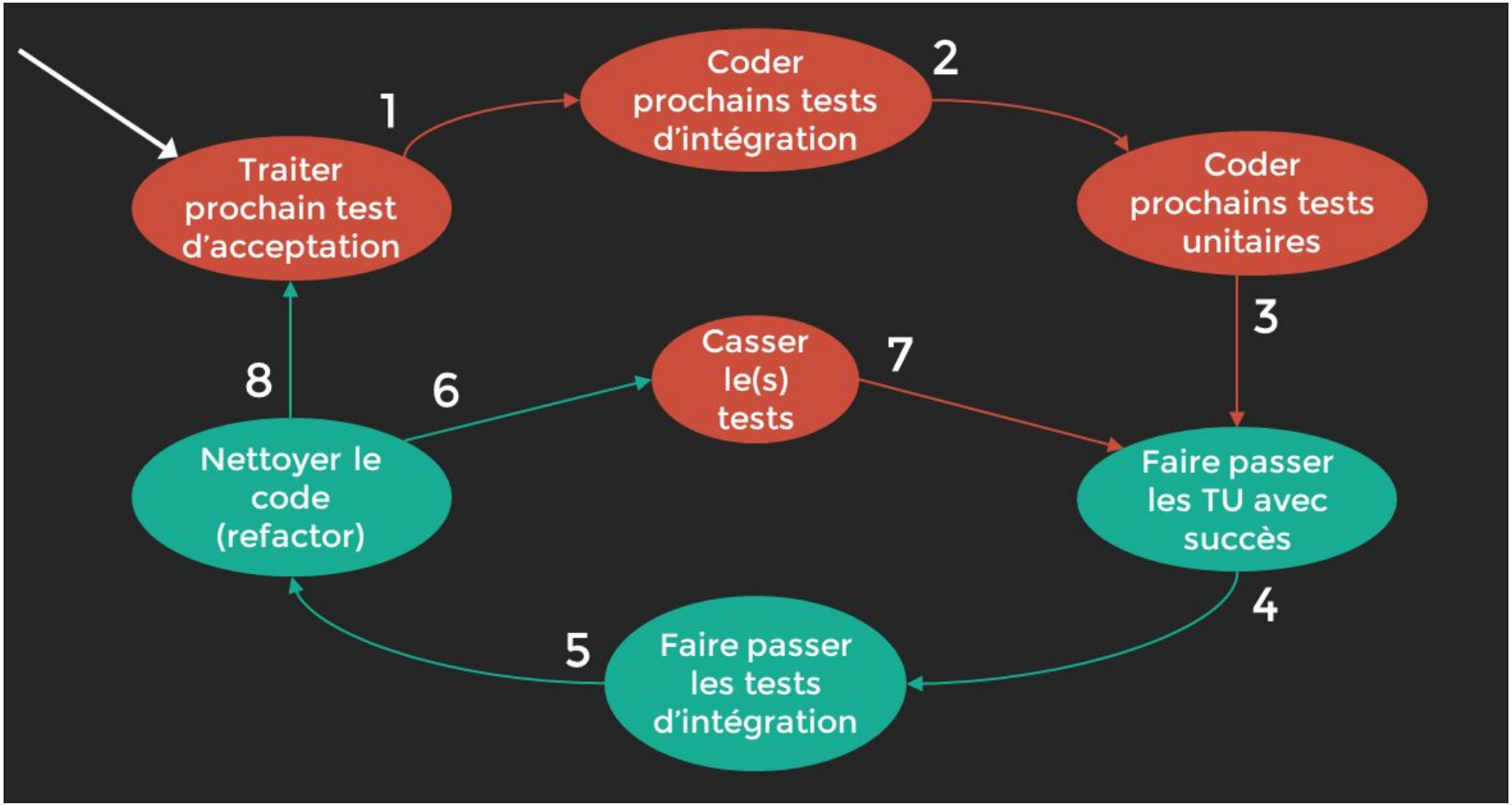


Focus : Integration Tests

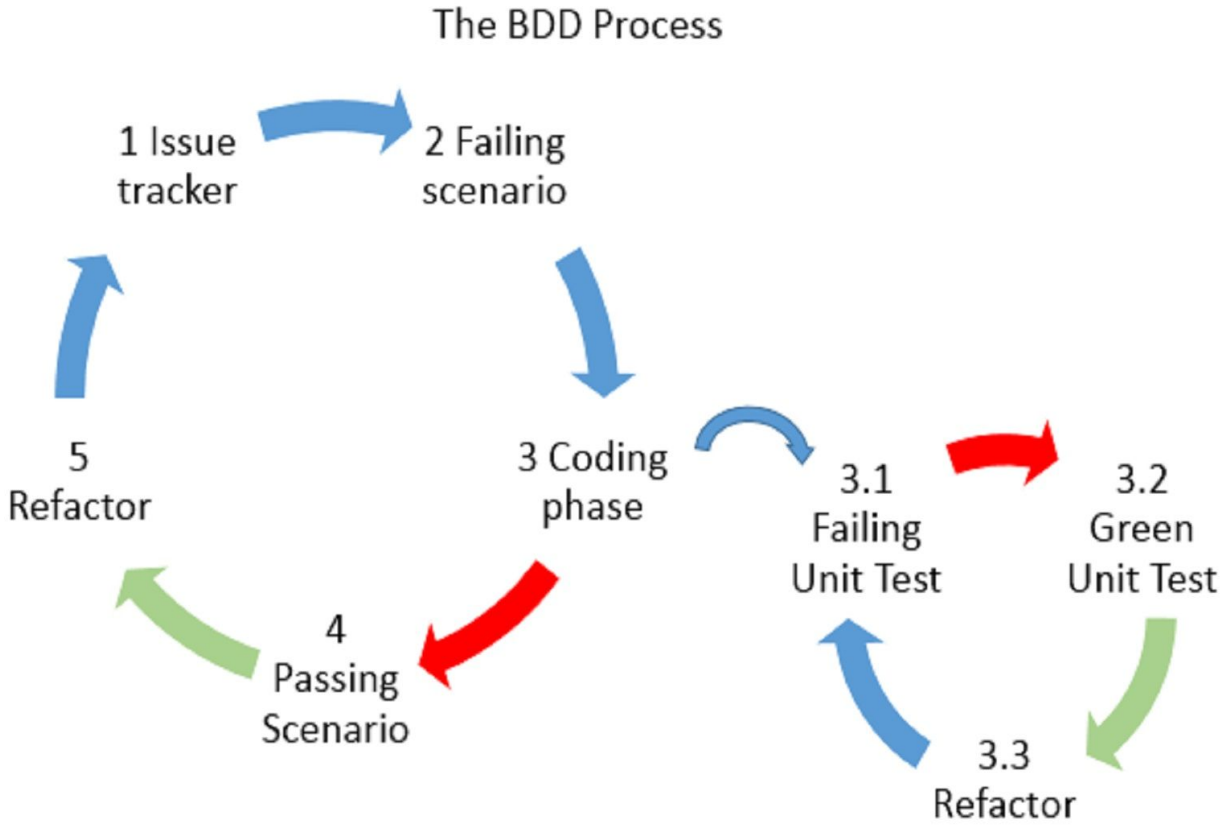
God save the queen … parce qu’ils sont différents

history ...

London TDD


Traiter prochain test d'acceptation
1
Coder prochains tests d'intégration
2
Coder prochains tests unitaires
3
Faire passer les TU avec succès
4
Faire passer les tests d'intégration
5
Nettoyer le code (refactor)
6
Casser le(s) tests
7
8

BDD : Behaviour Driven Development



BDD : Spock Framework

Spock Framework

Okay … sum plz

- Le **développement piloté par le comportement** (ou BDD) implique de collaborer avec votre équipe, les product owners, et d'autres décideurs, pour écrire des **tests d'acceptation** business en utilisant un langage métier. Ils deviennent vos premiers tests qui échouent.
- Le **TDD de Londres** ou *test de **l'extérieur vers l'intérieur*** implique de prendre ces tests d'acceptation et d'appliquer le rouge-vert-refactor des tests d'intégration système vers l'intérieur, vers les tests unitaires et les classes nécessaires à les satisfaire.
- **Cucumber** utilise un fichier de fonctionnalité non technique contenant des scénarios de tests d'acceptation. Ces scénarios testent automatiquement en écrivant des définitions **d'étape** pour correspondre au code.

In The End

- pratiquer le TDD avec les tests unitaires et le TDD de Londres avec les tests d'acceptation ;
- utiliser JUnit 5 et la bibliothèque d'assertions AssertJ ;
- structurer un test unitaire selon les 3 étapes AAA ou Given/When/Then ;
- vérifier la couverture de vos tests et la qualité de votre code avec SonarCloud
- utiliser Mockito pour créer des mocks et obtenir des tests bien isolés (le principe F.I.R.S.T. !) ;
- écrire différents types de tests d'intégration ;;
- écrire les tests fonctionnels de bout en bout et exploiter le pattern des PageObject