TESTS et TDD

l'idée des tests est d'automatiser les tests , de créer du code qui va tester notre code, plutôt que de le faire manuellement.

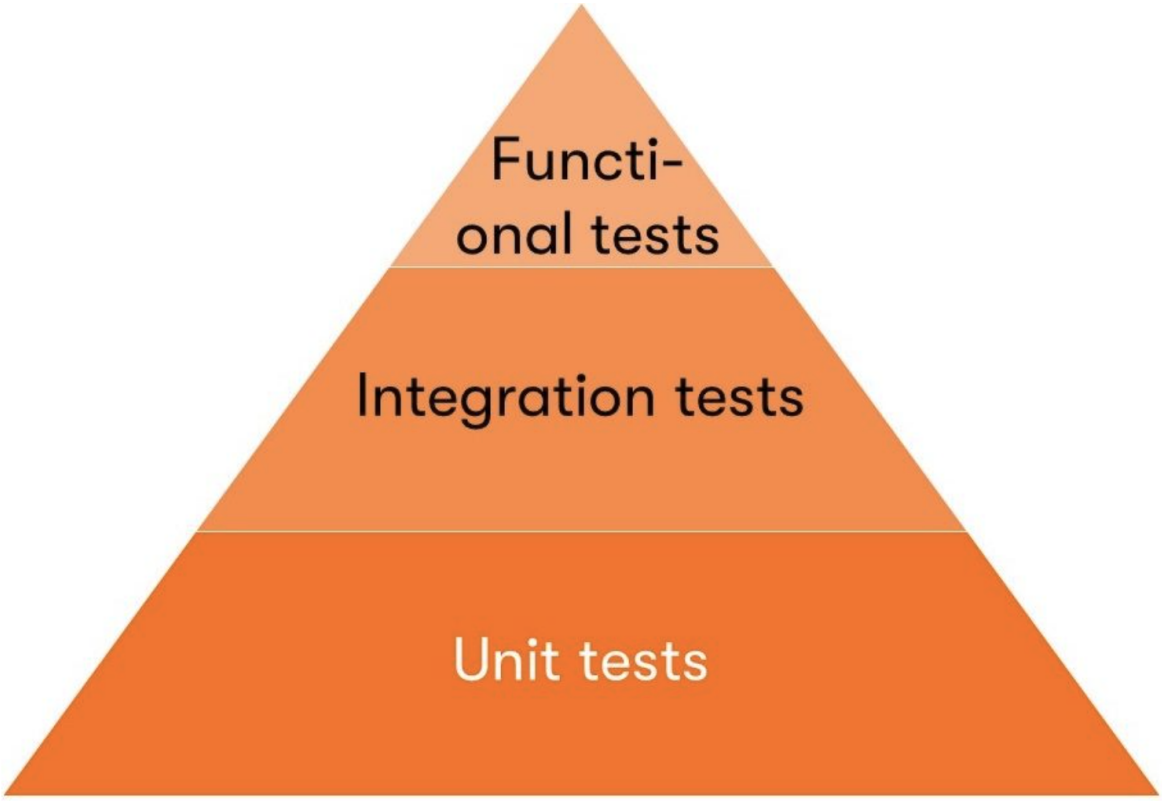
L'intérêt est qu'on va ainsi pouvoir envisager les éventuels bugs , qu'on va gagner du temps, que l'on va voir où sont les erreurs (parfois on modifie une variable qui va affecter une fonction plus loin dans notre code, on doit alors gérer des codes avec des dépendances complexes difficiles a tester manuellement car une fonction peut appeler plusieurs fonction). On va aussi pouvoir intégrer le code plus facilement dans un workflow (le code tester peut être déployer).

Les différents types de tests (unitaires, d'intégration, de parcours/validation)

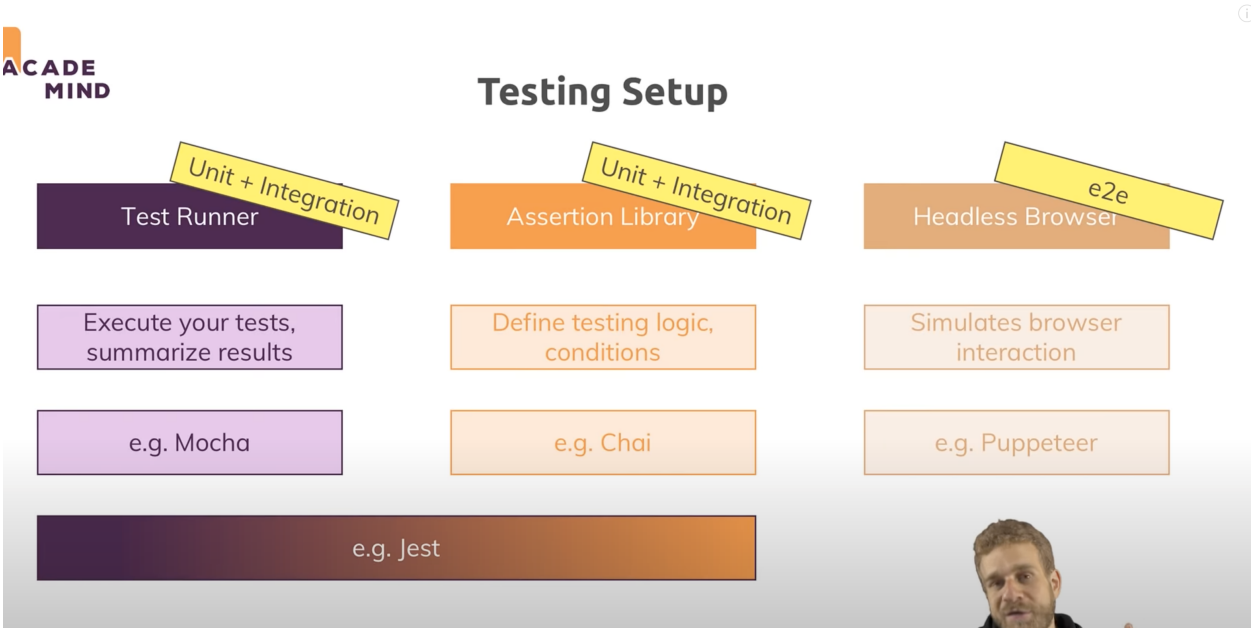
Les tests unitaires: Un test unitaire est un procédé permettant de s'assurer du bon fonctionnement d'une unité de programme. Le testing unitaire repose au fond sur un principe très simple : un système aussi gros et complexe qu'il soit est toujours composé différentes parties plus petites que lui. Il est indispensable (mais pas suffisant) que chacune de ces parties remplisse correctement sa fonction pour que l'ensemble soit fonctionnel et fiable.

Les tests d'intégration: Un test d'intégration vise à s'assurer du bon fonctionnement de la mise en œuvre conjointe de plusieurs unités de programme, testés unitairement au préalable.

Les test E2E (end to end), ou tests fonctionnels: on teste l'intégralité du code.



Testing set up en JS:



Choisissez un framework de test

Il existe plusieurs frameworks de tests, qui s'intègrent très bien avec les dernières versions de Visual Studio (même les versions gratuites).

Nous pouvons citer trois grandes options :

- Visual Studio Unit Testing Framework (appelé aussi historiquement MSTest), qui est le framework de Microsoft ;
- NUnit qui est un portage de JUnit, le framework de test pour Java ;
- xUnit.net, généralement abrégé en xUnit, qui est un framework plus récent, écrit par un des auteurs de NUnit.

le code coverage

La couverture de test (parfois appelée couverture de code, code coverage en anglais) est la mesure du taux de code source testé dans un système informatique.

La mesure de ce taux implique la réalisation de tests unitaires.

Les principales méthodes pour le calculer sont:

-couverture des fonctions, function coverage, (chaque fonction du programme a t elle été appelé?)

-couverture des instructions, statement coverage, (chaque ligne du code a t elle été exécutée et vérifiée?)

-couverture des points de tests, condition coverage, (les conditions/variables ont elles été testées?)

-couverture des chemins d'exécution, path coverage, (chaque chemin possible, par exemple true et false, a t il bien été exécuté et vérifié?)

popular coverage tools:

- Java: Atlassian Clover, Cobertura, JaCoCo
- Javascript: istanbul, Blanket.js
- PHP: PHPUnit
- Python: Coverage.py
- Ruby: SimpleCov

le TDD

Le test driven development (TDD) est une méthode de développement qui consiste à écrire son code de façon itérative et incrémentale, par petit bout, en écrivant chaque test avant d'écrire le code et remaniant le code continuellement.

Tandis que les procédures conventionnelles de test en aval utilisent un modèle en cascade ou en v, les processus du TDD sont cycliques. Cela signifie qu'il faut d'abord déterminer les cas tests qui échouent fréquemment. Cela est fait exprès,

car dans la deuxième étape, il ne faut écrire que la quantité de code nécessaire pour réussir le test. Les composants sont ensuite remaniés : tout en conservant ses fonctions, vous étendez le code source ou le restructurez si nécessaire.

Le Test Driven Development s'oriente vers les résultats des scénarios de test conçus par le développeur. La procédure cyclique garantit que le code n'est transféré au système de production que lorsque le logiciel répond à toutes les attentes. Cela signifie qu'il faut refaire et tester à nouveau les composants du code jusqu'à ce que le test réussisse. Cette méthode vous permet d'enrichir le logiciel de nouvelles fonctions étape par étape, car vous écrivez un nouveau code source après chaque test réussi. Pour cette raison, le TDD est également considéré comme l'un des modèles de développement incrémental dans le développement de logiciels.

Le Test Driven Development est une stratégie de conception, qui guide le processus de développement d'un logiciel au moyen de différents tests. Contrairement aux processus en aval, les scénarios de test de la méthode TDD sont effectués dès le début de la conception du logiciel. Les tests utilisés dans le contexte du TDD diffèrent par leurs objectifs et leurs portées.

au départ il s'agissait juste d'écrire les tests avant de coder puis **3 lois** ont émergé:

-on ne peut pas écrire le code tant qu'on a pas écrit un test qui échoue.

-on ne peut écrire qu'un test qui échoue à la fois (???)

-on écrit le minimum de code pour que le test en échec réussisse.

Le TDD a plusieurs **avantages**:

-éviter les modification inutiles du code. (on se focalise sur la satisfaction d'un besoin, étape par étape)

-éviter les les tests qui échouent sans qu'on puisse identifier l'origine du problème.

-maitriser le cout de la maintenance.

-s'approprier le code.

-fiabilité du code.

La méthode la plus connue pour le TDD est **la méthode du red/Green/Refactor:**

- **La phase rouge :** dans cette phase, il faut se mettre à la place de l'utilisateur. Celui-ci veut pouvoir utiliser simplement le code. Vous écrivez donc un test, qui contient des composants qui n'ont pas encore été implémentés. Vous devez décider quels éléments sont réellement indispensables pour que le code soit fonctionnel.
- **La phase verte** : on écrit un code le plus simple possible pour que le test réussisse.
- **Refactoring**: on améliore le code, on le restructure.

Avantages	Inconvénients
Le logiciel est de grande qualité et contient peu de bugs.	Nécessite la compréhension du code et exige une période de formation plus longue.
L'architecture du système et le code de production sont compréhensibles et bien structurés.	Teste uniquement l'exactitude du code et non la facilité d'utilisation du logiciel.
L'analyse des erreurs est rapide et les coûts de maintenance réduits.	Doit éventuellement être complété par d'autres procédures de test.
La suppression des redondances dans le code permet d'éviter la suringénierie.	

test unitaire (F.I.R.S.T.)

Les principes F.I.R.S.T sont utilisés pour écrire des tests unitaires de qualité.

-**F**ast : les tests doivent être rapides.

-**I**solé et Indépendant: tester un élément à la fois. Le test ne doit pas dépendre d'un autre test.

-**R**épétable: les tests doivent être autonomes et donner toujours les même résultats. Ils ne doivent pas dépendre des conditions (environnement de travail, jour, data etc)

-**S**elf-validation: Le test doit être capable de vérifier lui même les résultats automatiquement. On évitera les vérifications humaines qui peuvent devenir

rapidement chronophage et parfois amener à l'erreur (dû souvent à une lassitude de répéter régulièrement les mêmes tests).

-**T**horough: le code doit être testé largement pour tous les cas. Le test devra couvrir l'ensemble des cas de tests et non couvrir 100% des données. La raison est simple: en couvrant 100% des données, nous aurions des tests trop longs qui pourraient durer des jours à chaque exécution.

!!Attention au nommage des tests!!