METHODOLOGIE TESTS

TESTING


GOALS

PRODUCTION
DEVELOPMENT

QUE VA T'ON FAIRE ?

Evaluer les risques

Définir le niveau de qualité requis selon le projet

Définir les objectifs, les thèmes, les scénarios et les cas de test

Construire le plan de test

THÈMES DU PROGRAMME

PRODUCTION
DEVELOPMENT

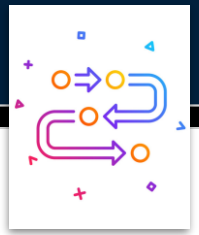
QUE VA T'ON FAIRE ? (suite)

TDD, qu'est ce que c'est

Avantages du TDD

Comment s'y préparer aux mieux

EVALUER LES RISQUES



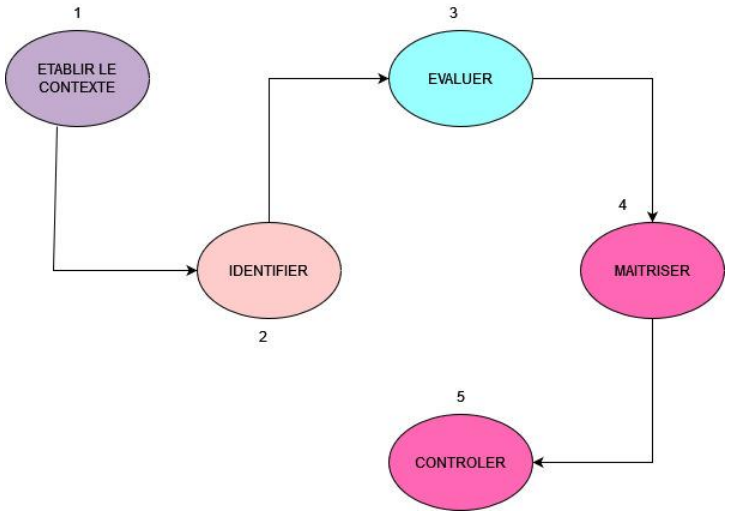
Qu’est-ce que cela veut dire ?

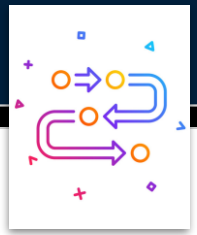
- Essayer d’anticiper les problèmes
- Permettre de comprendre les enjeux de chaque contraintes du projet
- Comprendre le projet
- Eviter aux maximums les pertes de temps
- Assimiler les tenants et aboutissants du projet
- Mise en place d’un registre de processus
- Amélioration continue du projet (pas que développement)

EVALUER LES RISQUES

Comment réaliser cela

- Avoir une matrice fiable pour répondre procéduralement à la question « quels sont les risques »



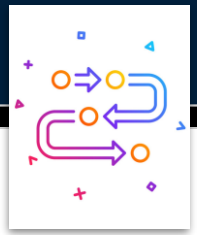


EVALUER LES RISQUES

ETABLIR LE CONTEXTE

- Effectuer en amont du développement
- Permettre de déterminer les contour du projet et donc du développement
- Interroger les participants au projet

EVALUER LES RISQUES



IDENTIFIER LES RISQUES

- Effectuer pour chaque problématique une analyse contextuelle

- Effectuer une matrice

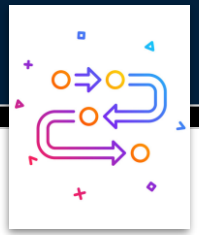
Exemple de questions :

•Cet événement est-il probable ?
•Quelle est la probabilité que ce risque survienne ?
•Si nous y sommes confrontés, quels seront son impact et sa gravité ?
•Quel est notre plan d'action en réponse au risque ?
•Compte tenu de sa probabilité et de son impact, quel est le degré de priorité ?
•Qui est responsable de la gestion de ce risque ?

EVALUER LES RISQUES

Table de gestion des risques ou Heat map

	1 Indolores	2 Limités	3 Graves	4 Dramatiques
1 Improbables				
2 Occasionnels				
3 Courants				
4 Très courants				



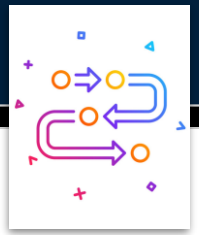
EVALUER LES RISQUES

EVALUER LES RISQUES

Donner un degrés d’ampleur

C’est là que vous aller responsabiliser vos tests

Définir quel partie du code sera tester par votest unitaire



EVALUER LES RISQUES

MAITRISE PAR VOS PROCESSUS DE TEST

Création des tests unitaires

Réaliser vos tests métier

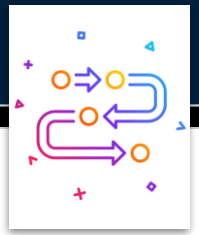
S’assurer que les bugs seront mineur

Pour répondre au mieux à ce genre de responsabilité il y a des moyens et des techniques de développement qui peuvent augmenter la fiabilité du projet

Par exemple il y a

TDD

EVALUER LES RISQUES



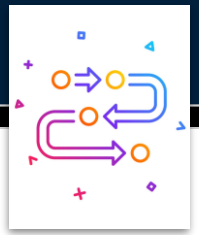
CONTROLER LES RISQUES

Réalisation des tests finaux

Corriger les Tests non effectifs

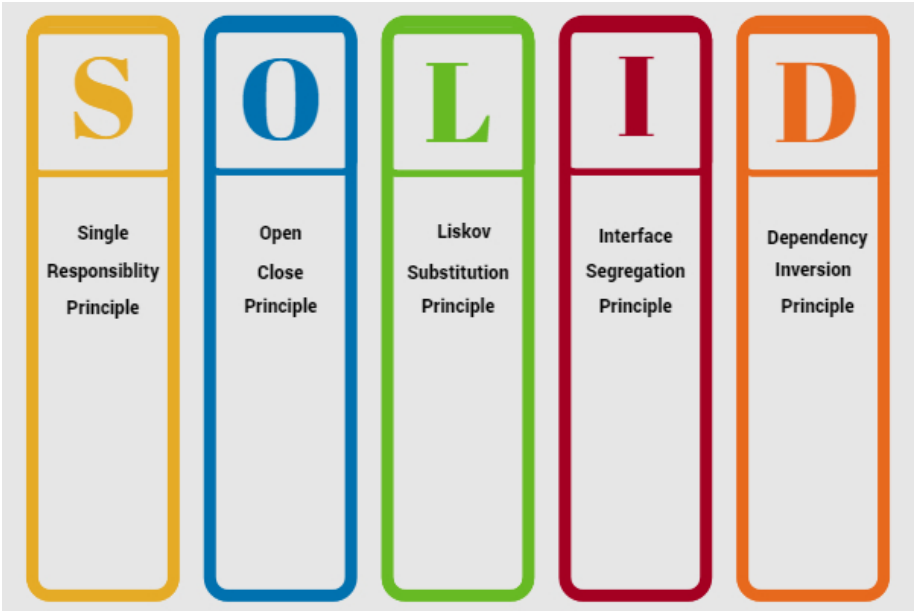
Refaire les Tests

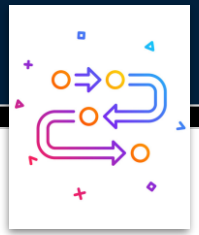
Fiabilité du projet



EVALUER LES RISQUES

Avant de parler TDD, on va faire un petit tour sur les principes SOLID





EVALUER LES RISQUES

S => Signle Reponsibility Principle (SRP)

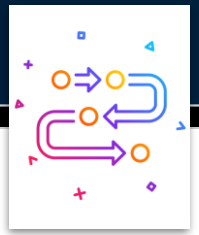
Pour faire simple c'est la responsibilité de code =>

« Une classe ne doit avoir qu'une seule et unique responsabilité »
Une machine à café est utilisé pour faire du café par pour faire du thé sinon on utlise une théiére

O => Open / Closed

Les entités doivent être ouvertes à l'extension et fermées à la modification
Cela signifie que l'extension du code est toujours préférable que la modification du code

Petit point :
Si tu commences à utiliser des « instanceof » dans des conditions , c'est que surement le piége s'est refermé sur toi



EVALUER LES RISQUES

L => LISKOV's Substitution Principle (LSP)

Pour faire simple c'est la responsabilité de code =>

Les objets doivent être remplaçables par des instances de leur sous-type sans pour autant altérer le fonctionnement du programme =>

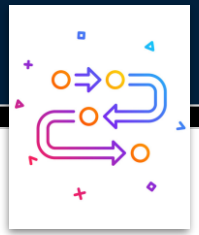
Signature des fonctions (paramètres et retour) doit être identique entre l'enfant et le parent

Les paramètres de la fonction de l'enfant ne peuvent pas être plus nombreux que ceux du parent

Le retour de la fonction doit retourner le même type que le parent

Les exceptions retournées doivent être les mêmes

En théorie en utilisant des classes abstract, le langage obligera de respecter les les principes SOL



EVALUER LES RISQUES

I => Interface Segregation Principle

Aucun client ne devrait être forcé d'implémenter des méthodes / fonctions qu'il n'utilise pas

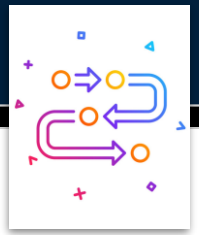
En résumé, il vaut mieux créer plusieurs petite interfaces qu'une seule grande

D => Dependency Inversion Principle (DIP)

Une classe doit prétendre de son abstraction pas de son implémentation

Autrement dit on évite de passer des objets en paramétre lorsqu'une interface est disponible.

Passer en paramètre une interface permet d'être certain que l'objet que tu manipules, peu importe son type, aura les bonnes méthodes associées.
Comme tu te poses la question je te réponds :
Non il n'y a aucun mal à passer des objets en paramètres de tes fonctions.
Ce principe s'applique surtout quand tu as une action commune à exercer pour plusieurs objets différents !



EVALUER LES RISQUES

TDD => Test Driven Developpement

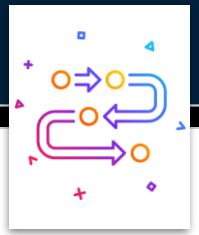
Principe de base :

Faire en sorte que le code non testable soit mineure ou devienne testable

Les tests sont à la base de tout

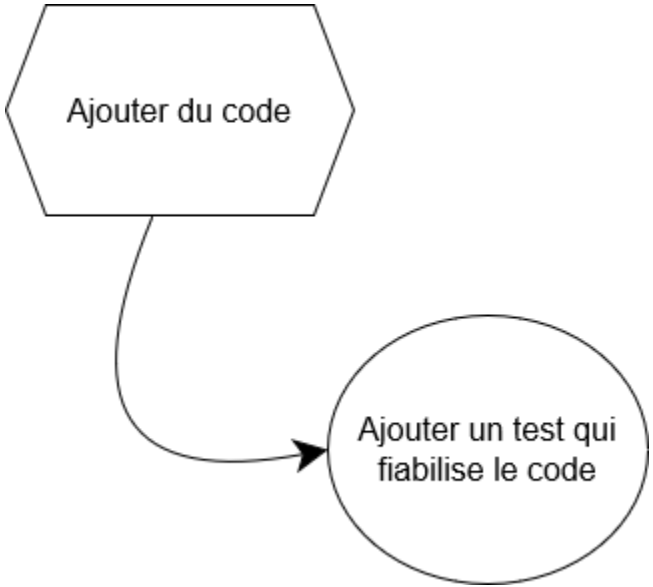
…

VOILA, C’est tout ce qu’est le TDD

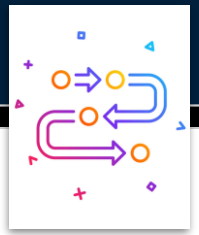


EVALUER LES RISQUES

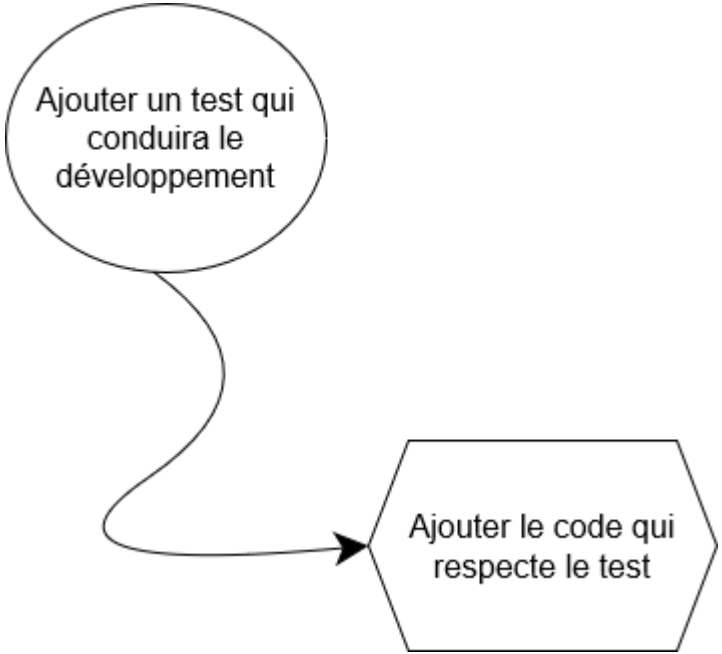
La métode de base est la suivante

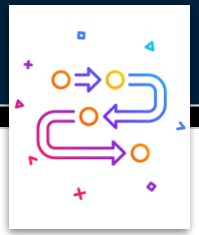


EVALUER LES RISQUES



Principe TDD





EVALUER LES RISQUES

Principe TDD : Démarche à suivre

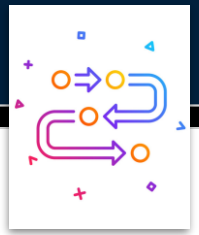
Décomposé en RGR =>

Les deux premières phases sont nommées d'après la couleur de la barre de progression dans les outils de test unitaires comme JUnit(**Red** pour échec et **Green** pour réussite)

R (Red) : Ecrire un code de test et les faire échouer

G (Green) : Ecrire le code métier qui valide le test

R (Refactor) : Remaniement du code afin d'en améliorer la qualité



EVALUER LES RISQUES

Cycle de développement

Le cycle de développement préconisé par TDD comporte **cinq étapes** :
Ecriture d'un premier test

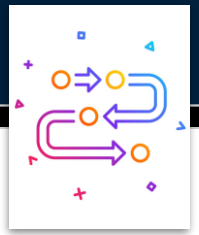
•Exécuter le test et vérifier qu'il échoue (car le code qu'il teste n'a pas encore été implémenté)

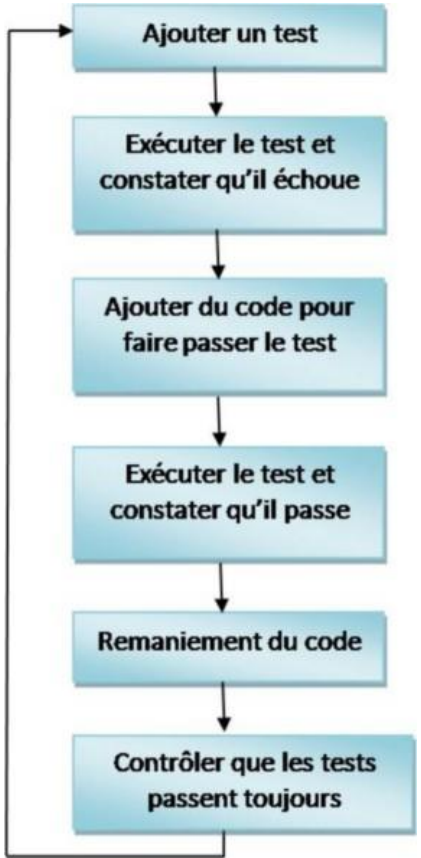
•Ecriture de l'implémentation pour faire passer le test (il existe différentes manières de corriger ce code)

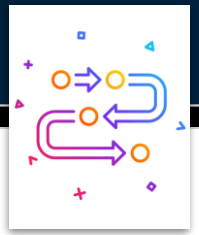
•Exécution des tests afin de contrôler que les tests passent et dans ce présent l'implémentation respectera les règles fonctionnelles des tests unitaires

•Remaniement (Refractor) du code afin d'en améliorer la qualité mais en conservant les mêmes fonctionnalités

EVALUER LES RISQUES







EVALUER LES RISQUES

Avantages

•Les tests unitaires sont réellement écrits

•La satisfaction du développeur permet d'obtenir un code plus cohérent

•Clarification des détails de l'interface et du comportement

•Vérification démontrable, répétable et automatisé

•Non présence de régression