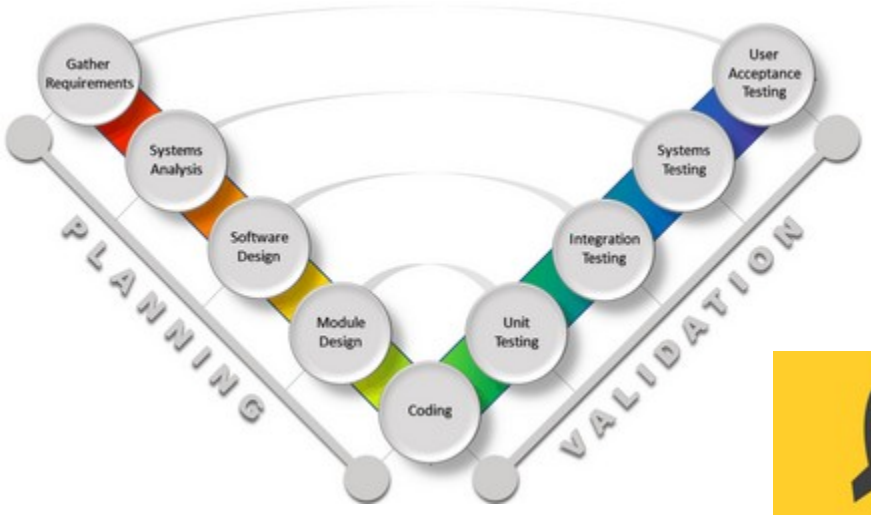
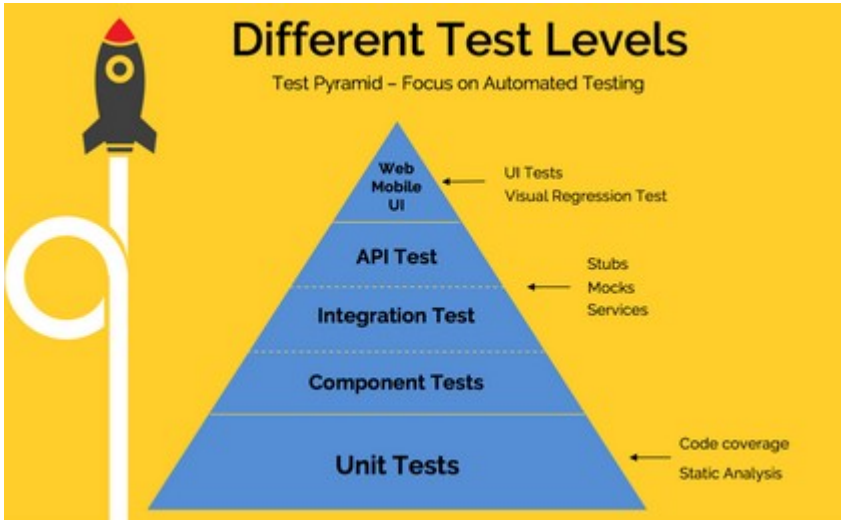
Tests et recette d’un projet informatique





- Introduction à cette formation
  - Votre formateur .. Et Vous
  - Le matériel
    - Outils de développement
    - Le cours
  - L'organisation – horaires
  - La forme :
    - Un mélange de concepts avec application directe par un exemple simple
    - Des exercices

Tests logiciels Sommaire

- Utilité des Tests
- Cycles de développement et Tests
- Coût des tests
- Présentation du Test-Driven Development
- Traitement des dépendances
- Tests et base de Données
- Des outils d’intégration continue

Tests logiciels

- Les liens utiles

  - https://openclassrooms.com/fr/courses/5641591-testez-votre-application-c
  - https://ibmcloud.developpez.com/cours/devops-pour-nuls/#LII-C-1
  - https://openclassrooms.com/fr/courses/7365126-concevez-une-strategie-de-test

  - https://sites.lesia.obspm.fr/emmanuel-grolleau/files/2016/07/Master_OSAE_Cours_Tests_Grolleau.pdf

- Assurer la fiabilité du logiciel est une part cruciale du développement
  Les tests représentent :

  - Plus de 50% du développement d'un logiciel critique
  - Plus de 30% du développement d'un logiciel standard

- Plusieurs études ont été conduites et indiquent que :

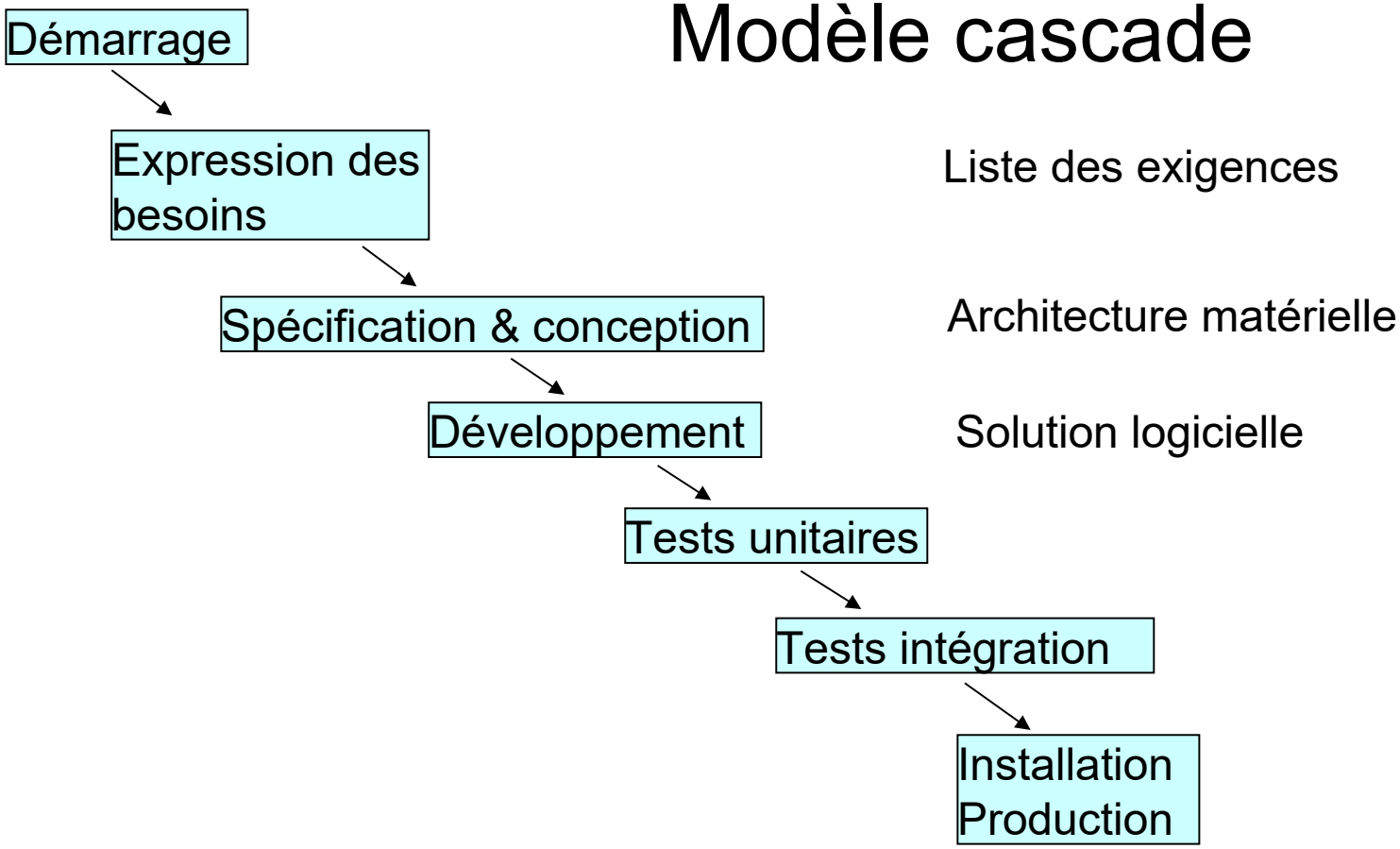
  - Entre 30 et 85 erreurs sont introduites par portion de 1000 lignes de code produites (logiciel standard).
  - Ces chiffres peuvent fortement augmenter si les méthodes permettant d'assurer la fiabilité sont mal gérées au sein du projet

Tests logiciels

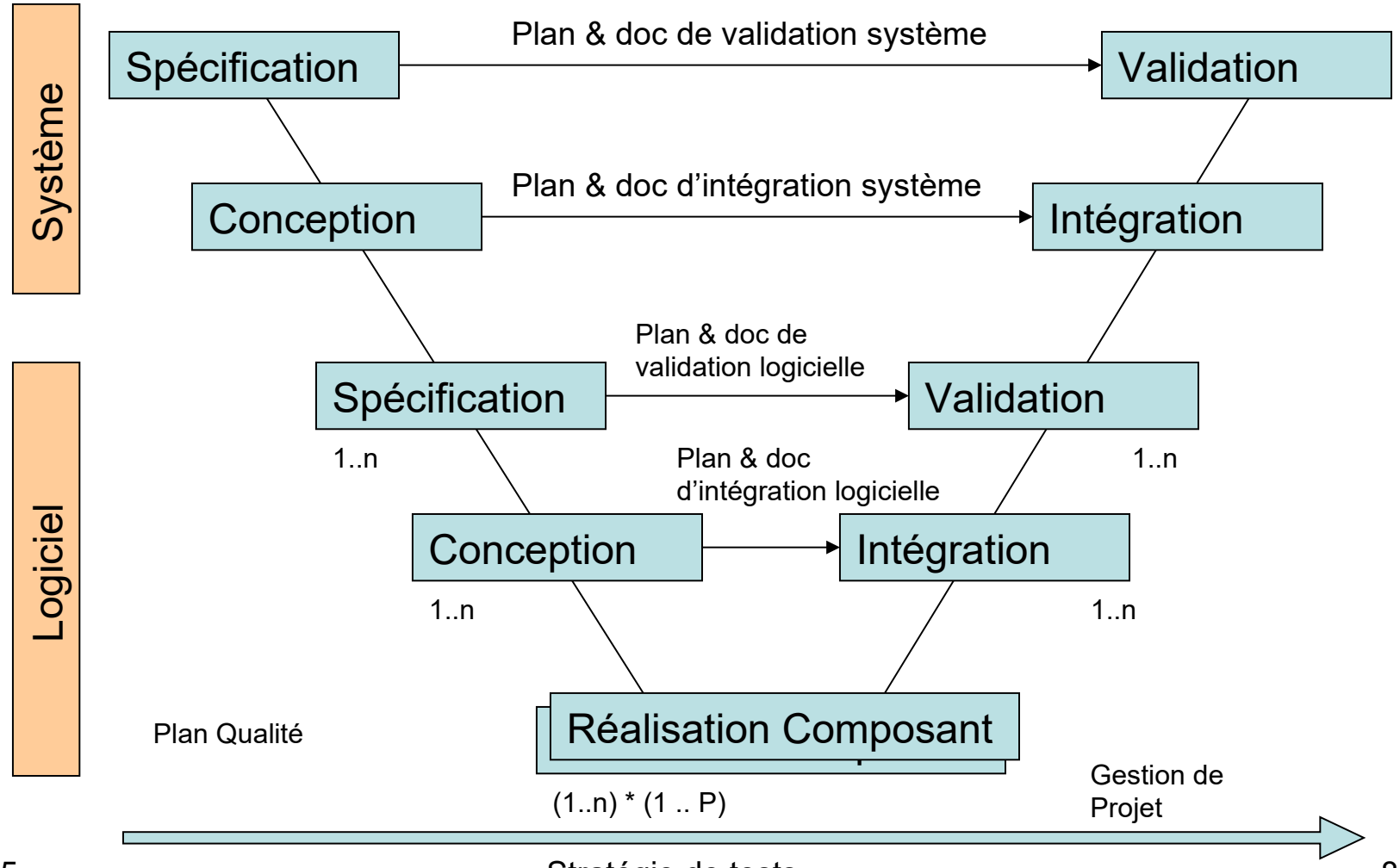
Le Comité Français des Tests Logiciels identifie quatre niveaux de tests :

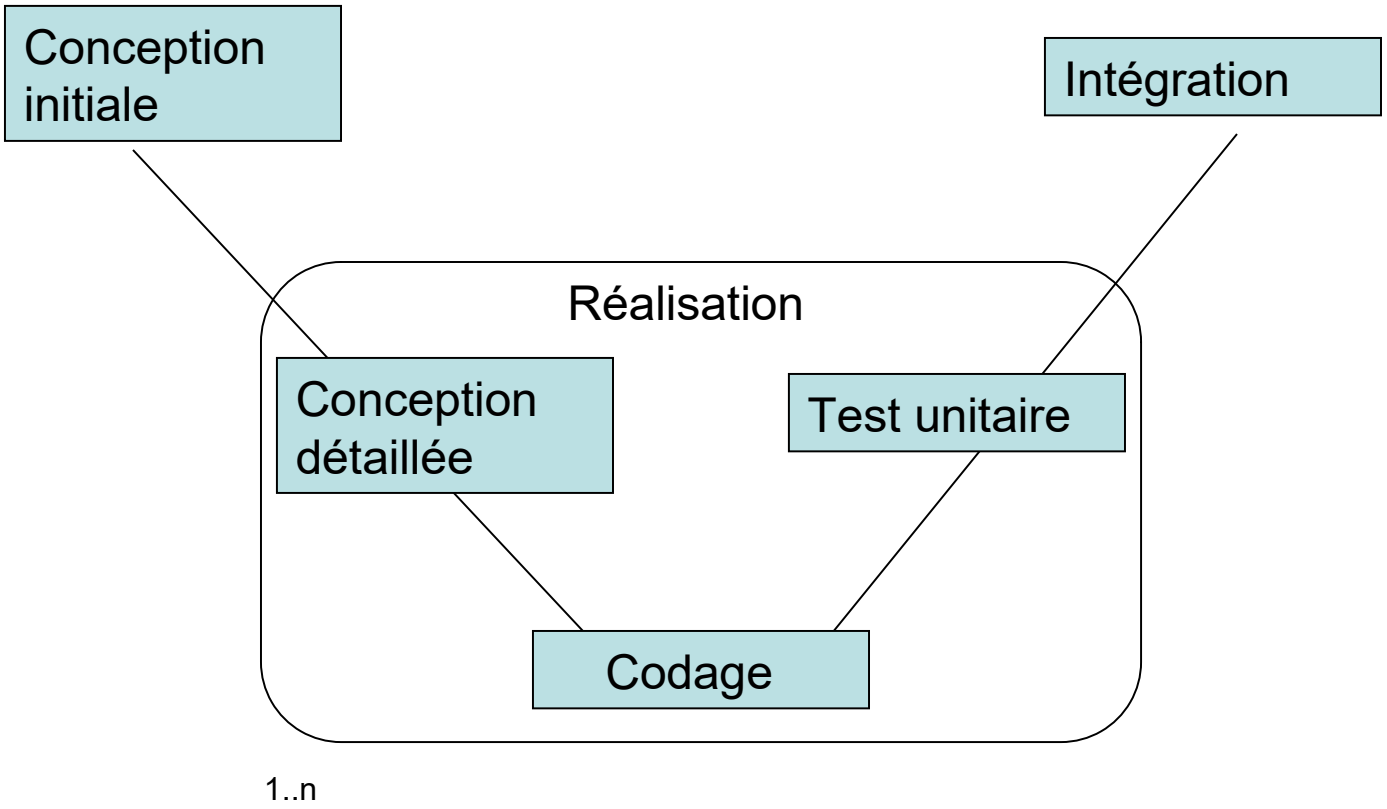
- Les tests unitaires permettent de tester un composant, ou un bout de code, isolé de ses dépendances.
- Les tests d'intégration permettent de vérifier que tous les bouts de code isolés fonctionnent bien ensemble.
- Les tests système permettent de tester l'application toute entière, dans les conditions proches du matériel client.
  Aussi appelé Validation
- Et enfin, les tests d'acceptation permettent de s'assurer que l'application répond bien au besoin fonctionnel.
  Aussi appelé Recette client

Modèle cascade



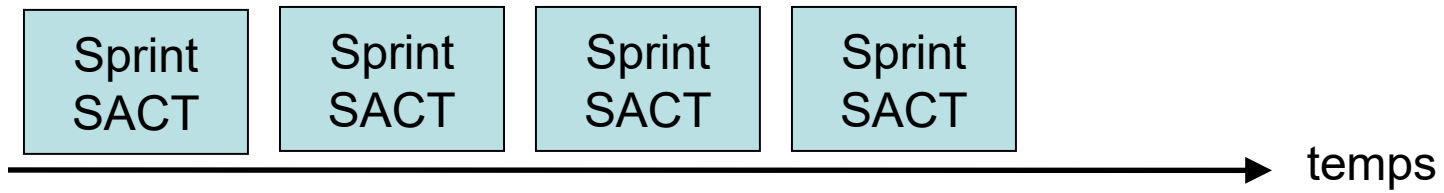
Cycle en V


Système
Logiciel
Spécification
Plan & doc de validation système
Validation
Conception
Plan & doc d'intégration système
Intégration
Spécification
Plan & doc de validation logicielle
Validation
1..n
Plan & doc d'intégration logicielle
1..n
Conception
Intégration
1..n
1..n
Réalisation Composant
Plan Qualité
(1..n) * (1 .. P)
Gestion de Projet


Conception initiale
Intégration
Réalisation
Conception détaillée
Test unitaire
Codage
1..n

Méthode Agile : Activités d'un Sprint

Scrum applique les bonnes méthodes pour chaque Sprint :
Spécification, Architecture (conception), Codage, Test : SACT

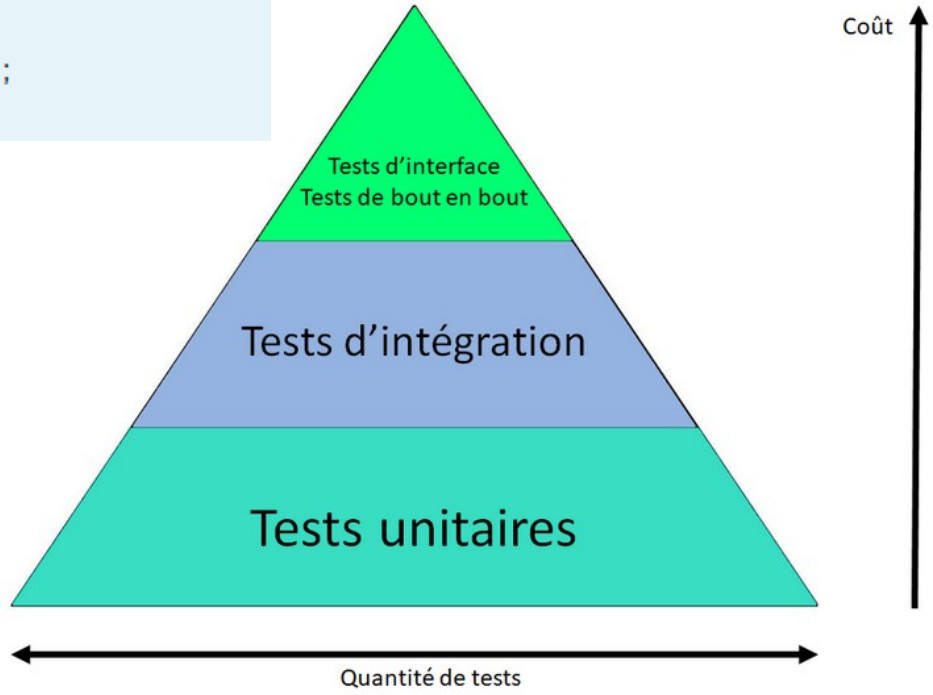


Tests logiciels Pyramide des tests

- Il faut trouver un équilibre entre les différents types de tests

Les ingénieurs de Google conseillent un ratio de :

- 70 % de tests unitaires ;
- 20 % de tests d'intégration ;
- 10 % de tests bout en bout.



Tests logiciels

- Principales méthodes :
  - Test statique
    - Revue des spécifications, des documents de design
    - Revue de code
    - Outil de vérification de code
  - Test dynamique
    - Exécuter le code par des tests manuels
    - Réaliser des tests automatiques

- Ce cours décrit les tests dynamiques automatiques

Tests logiciels

- Une pratique possible des tests : Le développement dirigé par les tests : TDD ou Test-Driven Development, un des concepts de eXtrem programming

- Le développement dirigé par les tests est une pratique basée sur

  - Les tests unitaires
  - L'écriture des tests par les développeurs
  - L'écriture des tests avant celle du code

Tests logiciels

L'écriture du code se fait suivant 6 itérations rapides

1. Conception rapide
2. Écriture d'un petit nombre de test unitaires **automatisés**
3. Lancement des tests pour vérifier qu'ils échouent (car pas encore de code correspondant)
4. Implantation du code à tester
5. Lancement des tests jusqu'à ce qu'ils passent tous
6. Restructuration du code et des tests si nécessaire pour améliorer les performances ou le design

- Un test doit être prédictif : on doit pouvoir décrire le résultat

- Un test doit être idempotent : le test doit pouvoir être rejoué autant de fois que nécessaire et fournir toujours le même résultat
  - Il faut maîtriser les dépendances (service web, contenu BD, fonction random …)
  - Remplacer des dépendances par des bouchons qui seront prédictibles

Les tests automatisés doivent être

- Concis – écrit aussi simplement que possible
- Auto-vérifiable – aucun intervention humaine ne doit être nécessaire
- Répétables – toujours sans intervention humaine (idempotent)
- Robustes – un test doit toujours produire le même résultat
- Suffisants – ils doivent couvrir tous les besoins du système
- Nécessaires – pas de redondances dans les tests
- Clairs – faciles à comprendre
- Spécifiques – en particulier chaque test d'échec pointe un point précis
- Indépendants – ne dépendent pas de l'ordre d'exécution des tests
- Maintenables – ils doivent pouvoir être aisément modifiés
- Traçables

Il faut donc bien admettre que le test exhaustif n'est pas possible
mais aussi que la sélection des tests est une activité complexe.

Le test est un processus destructif : un bon test est un test qui trouve
une erreur ;
alors que l'activité de programmation est un processus constructif : on cherche
à établir des résultats corrects

Les erreurs de codage peuvent être dues à des incompréhensions de
spécifications ou de mauvais choix d'implémentation

Vos efforts pour les tests devraient toujours être axés pour essayer de faire *planter* votre code

- Essayez des valeurs hors de la plage que vous avez spécifiée
- Essayez des quantités d'information qui sont nulles, en trop, et des quantités moyennes.
- Essayez des valeurs erronées (entrez une chaîne de caractères au lieu d'un nombre)…

Votre programme doit être écrit pour être testable

- Les modules et les méthodes doivent être petites et cohésives
- Si vous écrivez des méthodes qui sont complexes, vos tests vont être complexes
- Vous devez toujours penser à vos tests quand vous faites le design de vos programmes, car tôt ou tard vous allez avoir à tester ce que vous avez produit

Il y a des parties de votre code qui ne sont pas testables facilement :

- Les tests de défaillance, le traitement des erreurs et certaines conditions anormales
- Les fuites mémoire ne peuvent pas être testées. Vous devez inspecter votre code et vérifier que les blocs de mémoires sont libérés quand vous n'en avez plus de besoin (Sinon il faut utiliser des outils tels que purify)
- Les tests de performance

Les tests boîte noire ou boîte blanche :

- Le code source à tester n'est pas visible dans le cas de tests boîte noire. Les testeurs se focalisent sur les fonctionnalités externes définies dans la spécification et conception. Ces tests sont aussi nommés tests fonctionnels

- Pour des tests boîte blanche, le code est disponible. Des tests sont orientés suite à la lecture du code. Aussi appelés tests structurels.

En général, un test se décompose en trois parties, suivant le schéma «AAA», « Arrange, Act, Assert », que l'on peut traduire en français par « Arranger, Agir, Auditer » :

- Arranger : il s'agit dans un premier temps de définir les objets, les variables nécessaires au bon fonctionnement de son test (initialiser les variables, initialiser les objets à passer en paramètres de la méthode à tester, etc.).

- Agir : ensuite, il s'agit d'exécuter l'action que l'on souhaite tester (en général, exécuter la méthode que l'on veut tester, etc.)

- Auditer : et enfin de vérifier que le résultat obtenu est conforme à nos attentes.

Tests logiciels Parties d'un test

Le module JUNIT d' Eclipse permet d'implémenter facilement un environnement de test

JUnit

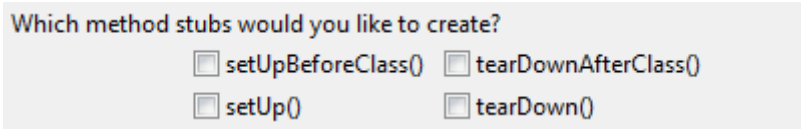
Nous allons réaliser une classe Calculation et la tester :
exercice 1

Dans l'exercice nous avons vu :

- La création d'un package de test. Il ne fera pas partie des livrables du projet
- Création d'un projet JUnit
- La définition de l'oracle en utilisant fail(), assertTrue(). D'autres fonctions existent (F1 dans eclipse).
- Le test de levée correcte d'une exception

Gestion du contexte

Des méthodes peuvent exister pour définir et initialiser un contexte de test, avant le passage des tests.



Elles sont proposées lors de la création du JUnit :

Annotation	Utilité
`@BeforeClass`	La méthode annotée sera lancée avant le premier test.
`@AfterClass`	La méthode annotée sera lancée après le dernier test.
`@Before`	La méthode annotée sera lancée avant chaque test.
`@After`	La méthode annotée sera lancée après chaque test.

Tester des méthodes qui n'ont pas de valeur de retour :

- Une telle méthode change alors un contexte, sinon elle ne sert à rien

- Le test consiste donc à tester le contexte, utiliser la méthode, puis tester l'évolution du contexte.
  (ex : ajout d'un élément dans une liste)

Couverture du code .

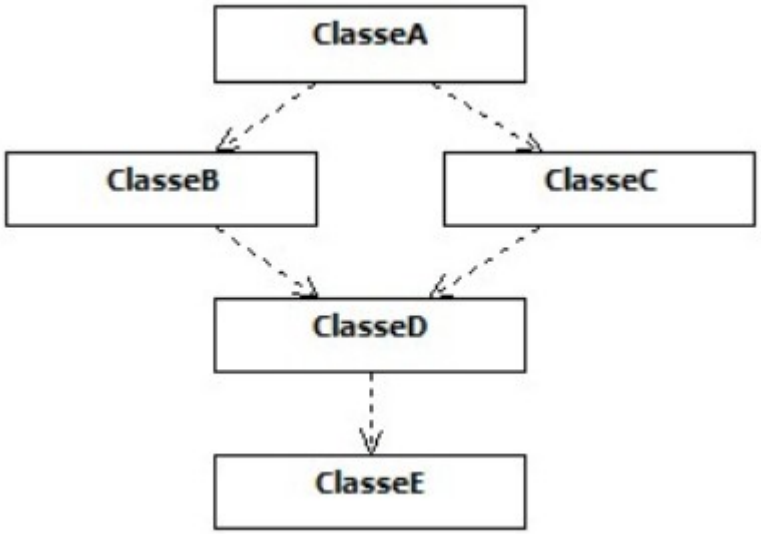
Nous allons utiliser le module EclEmma ou Jacoco : Exercice 2

- Dans une conception Objet les Objets sont en relation avec d'autres objets (dependancy, association, composition, agrégation ...)
- Si l'on teste plusieurs classes en même temps on réalise alors un test d'intégration et non pas un test unitaire
- Pour effectuer une test unitaire il faut isoler la classe à tester en simulant le comportement des autres classes : il faut réaliser des bouchons (stub)
- Les mock (simulacres, doublures) sont des bouchons dont le comportement est maîtrisé et prédictible (pour des tests idempotents)

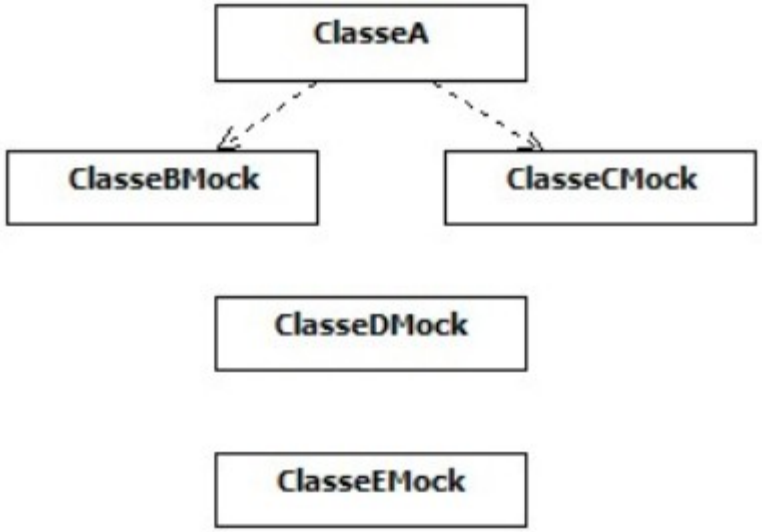
- Il existe deux types de mock :
  - Statique : c'est le développeur qui écrit des classes de substitution
  - Dynamique : le mock est mis en œuvre par un framework.

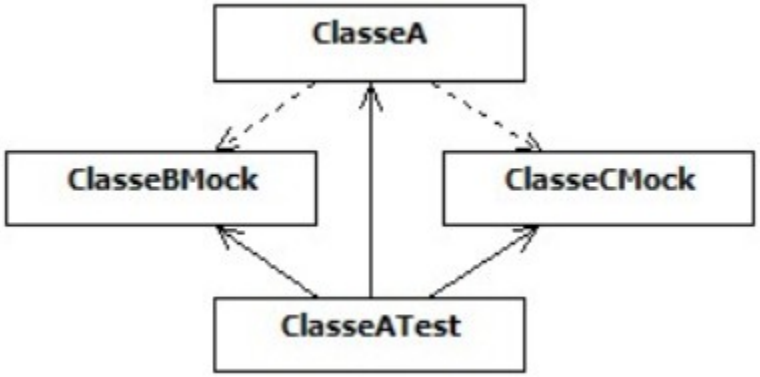
(se référer au site https://www.jmdoudoux.fr/java/dej/chap-objets-mock.htm)

- Il faut organiser son code pour qu'il soit testable sans modification
- Exemple : on veut faire un test unitaire de la classe A
-



- Pour les tests de A les mocks ClasseBMock et ClasseCMok vont simuler le fonctionnement de B et C



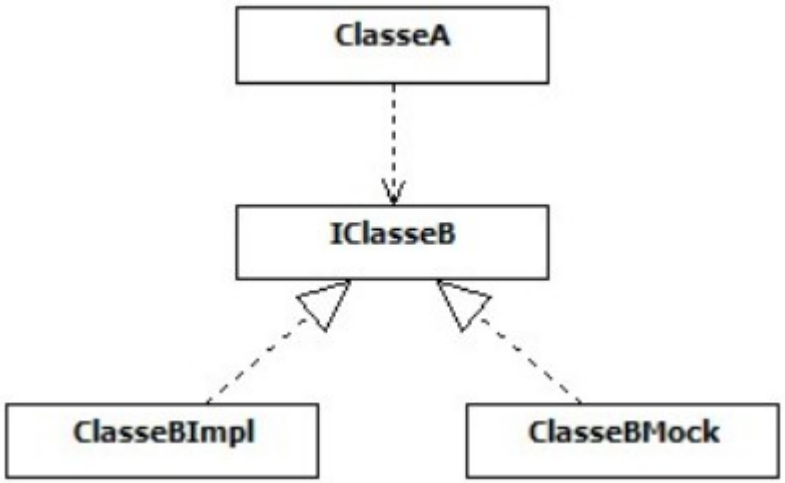

ClasseA
ClasseBMock
ClasseCMock
ClasseATest

L'entité testée ne doit pas savoir si l'objet utilisé durant les tests est un mock.

Pour cela, les dépendances d'une entité testée doivent être décrites avec une interface et l'entité doit utiliser un mécanisme d'injection pour permettre d'utiliser :

- une implémentation de la dépendance en production
- et d'utiliser un mock pour les tests.

Le fait de décrire les fonctionnalités d'une dépendance avec une interface facilite la mise œuvre d'un objet de type mock.



- Exercice 4 : réaliser un mock

Tests et base de Données

- Pour qu'un test soit rejoué à l'infini (idempotent), il ne doit pas modifier l'état du système.
  Tout le contraire avec une BD car son état évolue en permanence
- Comment alors effectuer des tests d'intégration avec uns BD ?
- Plusieurs pistes possibles :

1) Maîtriser la création et destruction de la BD
   Le test commence par créer la base ou la table
   Le/les test s'exécute
   Le test détruit la base ou la table
2) Utiliser un backup de la base de données
   Le test commence par mettre en place/restaurer une BD dans un état initial connu
   Le/les test s'exécute

- Plusieurs pistes possibles (suite):
  3) Code dans le test pour nettoyer la BD après l'exécution
  Attention : maîtriser les tests qui ont échoué

  4) Code dans le test pour connaître l'état de la BD pour un certain contenu
  Passer les tests
  Code dans le test pour constater le/les changement
  Concevable si l'absence de nettoyage n'est pas nuisible aux autres tests

Il existe des environnements de travail permettant de lancer automatiquement des tests :

- GitLab et particulièrement GitLab ci https://www.youtube.com/watch?v=URcMBXjIr24

- Jenkins https://www.youtube.com/watch?v=Gy4Nk2pIuNs