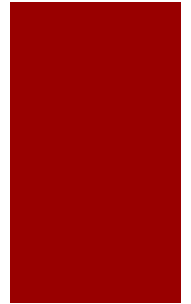




Test

Anne Etien

Motivations

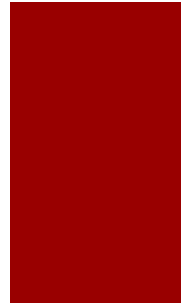


- Coût des bugs
 - Coûts économique : 64 milliards \$/an rien qu'aux US (2002)
 - Coûts humains, environnementaux, etc.
- Nécessité d'assurer la qualité des logiciels
- Domaines critiques
 - atteindre le (très haut) niveau de qualité imposée par les lois/normes/assurances/... (ex : DO-178B pour aviation)
- Autres domaines
 - atteindre le rapport qualité/prix jugé optimal (c.f. attentes du client)

Validation et Vérification

- **V**alidation et **V**érification (V & V)
- Vérification :
 - est-ce que le logiciel fonctionne correctement ?
 - “are we building the product right ?”
- Validation :
 - est-ce que le logiciel fait ce que le client veut ?
 - “are we building the right product ?”

Validation et Vérification



■ Quelles méthodes ?

- Revues de code
- simulation/ animation
- tests (méthode de loin la plus utilisée)
- méthodes formelles (encore très confidentielles, même en syst. Critiques)

■ Coût de la V & V

- 10 milliards \$/an en tests rien qu'aux US
- plus de 50% du développement d'un logiciel critique (parfois > 90%)
- en moyenne 30% du développement d'un logiciel standard

Définition

Le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus

IEEE-STD729, 1983

Développement d'un test

1. choisir un cas de tests = scénario à exécuter
2. estimer le résultat attendu du cas de test (Oracle)
3. déterminer (1) une donnée de test suivant le cas de test, et (2) son oracle concret (concrétisation)
4. exécuter le programme sur la donnée de test (script de test)
5. comparer le résultat obtenu au résultat attendu (verdict: pass/fail)

■ Script de Test :

- code / script qui lance le programme à tester sur le DT choisi, observe les résultats, calcule le verdict

■ Suite / Jeu de tests :

- ensemble de cas de tests

Cas de test

- Le test ne peut pas prouver au sens formel la validité d'un programme
 - Testing can only reveal the presence of errors but never their absence.

E. W. Dijkstra (Notes on Structured Programming, 1972)

- Un bon jeu de tests doit donc :
 - exercer un maximum de “comportements différents” du programme (notion de critères de test)
 - tests nominaux : cas de fonctionnement les plus fréquents
 - tests de robustesse : cas limites / délicats

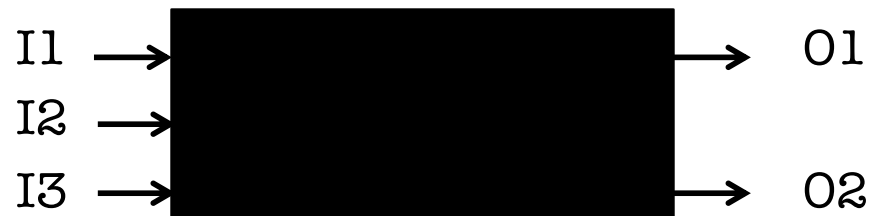
Intérêt du test

- Se poser la question du résultat de l'exécution
 - Définir un oracle impose de connaître le résultat
- Exécuter son propre code (ou celui d'un autre)
 - Être son premier utilisateur
 - S'interroger sur la pertinence de l'interface des classes
- Réfléchir aux cas limites
 - Identifier les cas « normaux » = simple
 - Identifier les cas « limites » = compliqué mais très utile
- Non régression
 - En cas d'évolution du code, le comportement n'a pas changé

Catégorie de tests

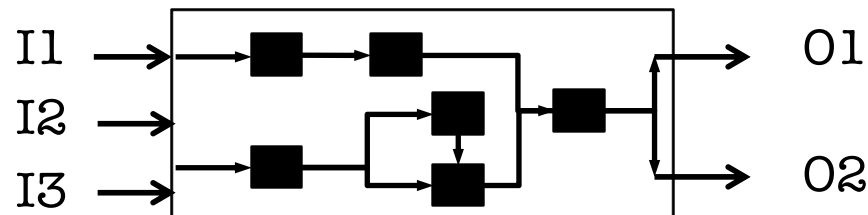
■ Test en boîte noire

- Utilise la description des fonctionnalités du programme
- Les test en boîte noire s'exécutent en ignorant les mécanismes internes (sans accès au code source par le testeur)

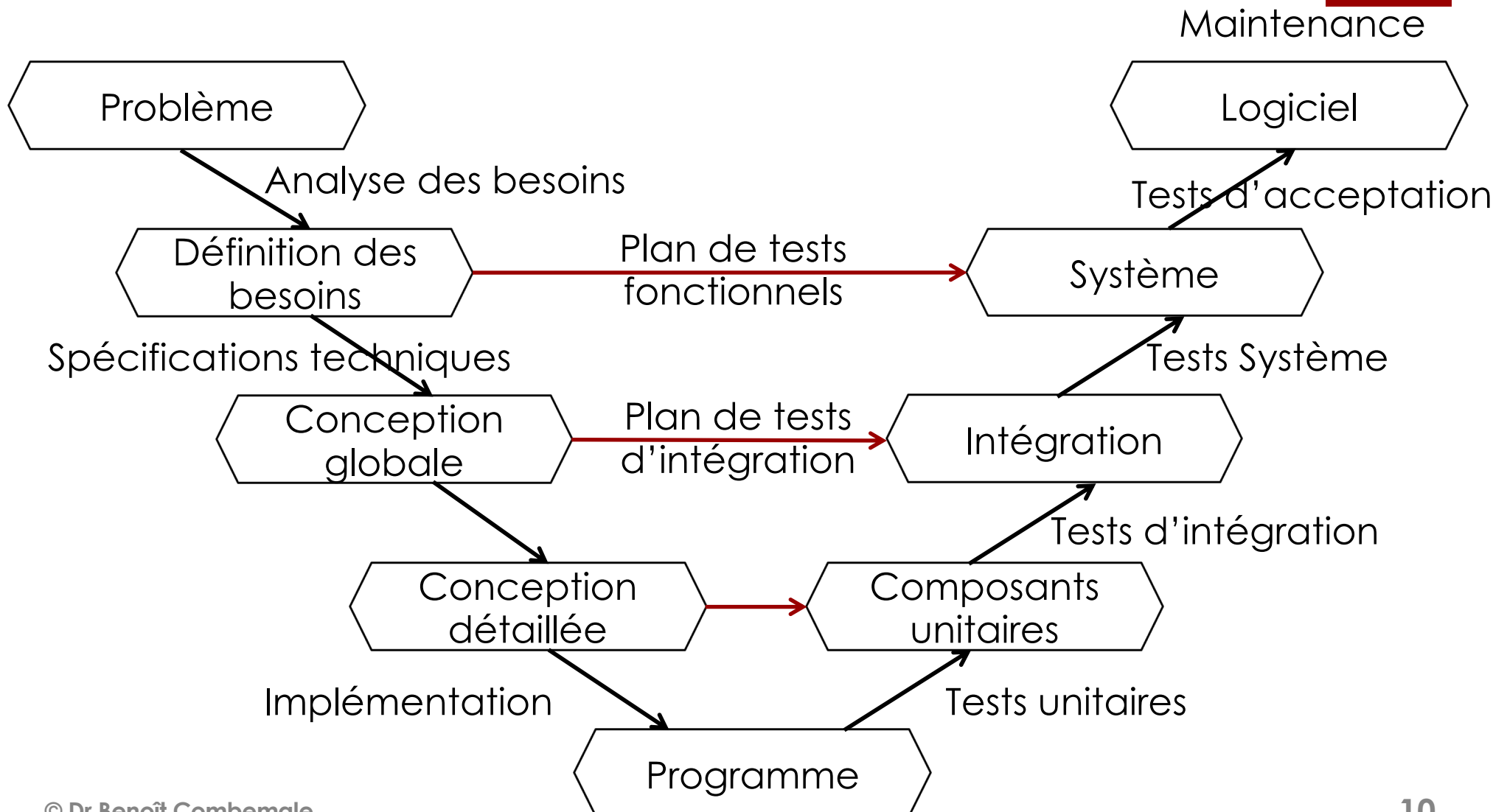


■ Test en boîte blanche

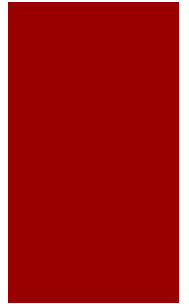
- Utilise la structure interne du programme



Hiérarchie des tests



Tests Unitaires



- Tests de blocs individuels (par exemple des blocs de code)
 - Exécuté en boîte blanche
- Généralement écrits par les développeurs eux-mêmes pour la validation de leurs classes et leurs méthodes (ou leurs fonctions en procédural)
- Généralement exécutés par les machines

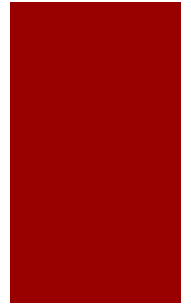
Tests d'intégration

- Exécuté en boîte noire ou boîte blanche
- Vérifient que les composants s'intègrent bien avec d'autres composants ou systèmes
- Vérifient aussi que le produit est compatible avec l'environnement logiciel et matériel du client


Tests fonctionnels

- S'exécute en boîte noire
- Vérifie que le produit assure les fonctionnalités spécifiées dans les spécifications fonctionnelles

Tests système



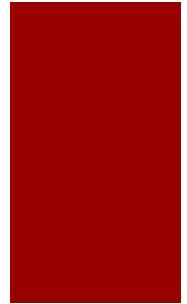
- S'exécute en boîte noire
- S'oriente vers les spécifications non fonctionnelles
- Composé de plusieurs tests :
 - Tests de charge : tester le produit au-delà des attentes non fonctionnelles du client. Par exemple, un système de sauvegarde qui doit tout sauvegarder deux fois par jour, le tester en mode trois fois par jour.
 - Tests de performance : évaluation par rapport à des exigences de performances données. Par exemple, un moteur de recherche doit renvoyer des résultats en moins de 30 millisecondes.

- 
- Tests de chargement : vérifie que l'application fonctionne dans des conditions normales (contraire aux tests de stress)
 - Tests d'utilisabilité : vérifier les exigences d'apprentissage demandées à un utilisateur normal pour pouvoir utiliser le produit

Test d'acceptation

- S'exécute en boîte noire
- tests formalisés par le client correspondant à ses exigences accepter le produit

Comparaison entre les tests



Tests	Portée	Catégorie	Exécutant
Unitaires	Petites portions du code source	Boîte blanche	Développeur Machine
Intégration	Classes / Composants	Blanche / noire	Développeur
Fonctionnel	Produit	Boîte noire	Testeur
Système	Produit / Environnement simulé	Boîte noire	Testeur
Acceptation	Produit / Environnement réel	Boîte noire	Client



Tests unitaires

Unité de test

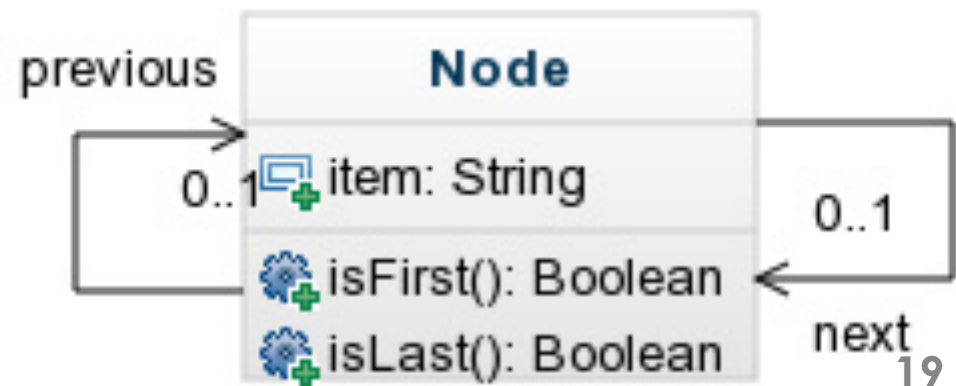
- Pour un langage procédural

- Unité de test = procédure

```
void Ouvrir (char *nom, Compte *C, float S, float D)
{
    C->titulaire = AlloueEtCopieNomTitulaire(nom);
    (*C).montant = S ;
    (*C).seuil = D ;
    (*C).etat = DEJA_OUVERT ;
    (*C).histoire.nbop = 0;
    EnregistrerOperation(C);
    EcrireTexte("Ouverture du compte numero ");
    EcrireEntier(NumeroCourant+1);
}
```

- Pour un langage OO

- Unité de test = classe



Objectifs

- Détecter le maximum de défaillance avant les tests en boîte noire et qu'ils peuvent s'exécuter d'une manière automatique
- Deux sous objectifs : « couverture de code » et « couverture de données »
 - La couverture du code = tester chaque ligne de code écrite (appel de fonction, boucles, décisions, décisions multiples,...)
 - La couverture des données oriente les tests vers les données (données valides, données invalides, accès aux éléments en dehors de la capacité d'un tableau, peu de données, trop de données,...etc)

Caractéristiques

- Les tests unitaires doivent être conforme à l'acronyme FIRST:
 - FAST : un test unitaire doit s'exécuter rapidement
 - INDEPENDANT: chaque test doit être indépendant des autres
 - REPEATABLE: chaque test peut être répété autant de fois que voulu
 - SELF-VALIDATING: Un test se valide lui-même par un succès ou par un échec
 - TIMELY: On écrit les tests quand on en a besoin

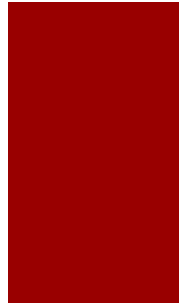


JUnit

JUnit

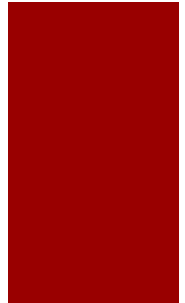
- www.junit.org
- framework de test unitaire pour le langage de programmation Java.
- Créé par Kent Beck et Erich Gamma,
- projet de la série des xUnit connaissant le plus de succès.

JUnit Test instructions



Instruction	Description
<code>fail(String)</code>	Make fail the test method
<code>assertTrue(true)</code>	Always true
<code>assertEquals(expected, actual)</code>	Test if the values are the same
<code>assertEquals(expected, actual, tolerance)</code>	Proximity test with tolerance
<code>assertNull(object)</code>	Check if the object is null
<code>assertNotNull(object)</code>	Check if the object is not null
<code>assertSame(expected, actual)</code>	Check if the variables refer the same object
<code>assertNotSame(expected, actual)</code>	Check if the variables do not refer the same object
<code>assertTrue(boolean condition)</code>	Check if the boolean condition is true

JUnit Test annotations



Annotation	Description
@Test	Test method
@Before	Method executed <i>before each test</i>
@After	Method executed <i>after each test</i>
@BeforeClass	Method executed <i>before the first test</i>
@AfterClass	Method executed <i>after the last test</i>
@Ignore	Method not run as test

Annotations have to be put before the methods of the unitary test class

Test d'exception

- Il est possible de tester qu'une exception doit être provoquée
- Compléter l'annotation `@Test` de la méthode de test par:

`@Test(expected=ExceptionName.class)`

- Le test sera alors :
 - valide si l'exception `ExceptionName` est provoquée
 - invalide sinon.

Procédures de test

- à programmer dans de simples classes dites **classes de test**
- sous la forme de **méthodes annotées** par **@Test**
- exploitant le jeu d'**assertions** précédentes.
- les classes et méthodes de test doivent être **public** pour être visibles de JUnit, notamment de son moteur d'exécution (`org.junit.runner`)
- bonne pratique : les regrouper dans un package `test`

Jouer les tests

- dans un terminal (quelque soit l'outillage utilisé) moyennant l'ajout de la lib **junit.jar**
 - compiler les classes de test

```
javac
    -cp <path to junit.jar>
    test.CompteTest.java
```
 - exécuter

```
java
    -cp <path to junit.jar>
    org.junit.runner.JUnitCore
    test.CompteTest
```
- ou dans les outils (IDE) intégrant JUnit tels qu'Eclipse ...

Conclusion

- démarche itérative
 - écrire les tests progressivement et en même temps que le code (allers-retours)
- Possibilité de rejouer indéfiniment et sans effort après toute modification du code : correction, ajout de fonctionnalités, changement de choix d'implem (versions) ...
- « maximiser » les tests
 - en termes de couverture de code et de couverture de données
- Cas extrême le Test Driven Development