



Validation, vérification et tests

Master 1 MIMÉ 2016



Validation, vérification et tests

- **Introduction**
- **Validation**
- **Vérification**
- **Tests**

1 - *Introduction*

- L'*activité de **test*** d'un logiciel utilise
 - différents types et techniques de tests pour vérifier que le logiciel est **conforme** à son cahier des charges (**vérification** du produit) et **aux attentes du client** (**validation** du produit).
 - Elle est un des processus du développement de logiciels.

2 - Validation

- La validation consiste à vérifier une spécification par rapport aux **attentes métier**.
 - Par exemple, après la validation d'une facture, la quantité du produit vendu doit décroître.
- Réponse à la question :
 - est-ce que le produit fait ce qu'on lui a demandé et répond bien au besoin du client ?

3 - *Vérification*

- La vérification consiste à vérifier une spécification par rapport aux **attentes techniques**.
 - Par exemple, après la validation d'une facture, retrouver le champ « valide » à « true » dans la base
- Réponse à la question :
 - faisons-nous le produit **correctement** ?



Tests

4 - *Qu'est ce qu'un test? (1)*

- Le test est l'exécution ou l'évolution d'un systèmes ou d'un composant du système, par des moyens automatiques ou annuels
 - Pour vérifier qu'il répond à ses spécifications, ou
 - Pour identifier les différences entre les résultats attendus et les résultats obtenus »
 - IEEE STD 729 Standrd glossary of software engineering

C'est l'art et la manière de trouver des bugs

4 - Qu'est ce qu'un test? (2)

Condition de test

- Qu'est ce qu'on cherche à évoluer ? (Raison et l'objectif du test?) :
 - Tester quoi?
 - Et à quel moment?
 - Pourquoi tester?
- Qu'est ce qu'il faut examiner pour cela?
- Par apport à quoi ? (Quelle est la référence?)
- Qui teste?
- De quoi a-t-on besoin pour tester?
- Quel est le rapport coût/bénéfice de l'activité de test?

4 - Qu'est ce qu'un test? (3)

Cas de test

- C'est une instance détaillée de la condition de test qui définit:
 - Les données d'entrées
 - Les préconditions (conditions préalables nécessaires au démarrage du test)
 - Les post conditions (conditions assurées après l'exécution du test)
 - L'oracle, un processus fournissant les références et permettant de prononcer le verdict du test
 - Le résultat attendu

4 - Qu'est ce qu'un test? (4)

L'oracle

- un composant soumis aux mêmes actions que l'application à tester
 - qui servira à définir si le résultat retourné par celui-ci est valide ou non.
 - peut être un logiciel très simple qui implémente les mêmes algorithmes.
 - Il y a plusieurs types d'oracles:
 - exhaustif: l'algorithme est implémenté de manière à produire les mêmes résultats.

4 - Qu'est ce qu'un test? (5)

Procédure de test

- Une réalisation pratique du cas de test
 - Automatique ou manuelle

- Séquence d'actions pour l'exécution du test
 - Récupération des données d'entrée
 - Constructions des préfabriqués
 - Exécutions des opérations
 - Récupération des verdicts
 - Nettoyage

4 - Qu'est ce qu'un test? (6)

Verdict

- Réponse à une requête d'exécution d'un test
 - ☐ OK : réussit
 - ☐ KO : échec
 - ☐ NA : Non appliqué
 - ☐ NE : non exécuté
 - ☐ NI : non implémenté

4 - Qu'est ce qu'un test? (7)

aspects du test

- Le test est de trouver des défauts et pas seulement de montrer que « ca fonctionne »
 - Exception : les tests d'acceptation
- Les développeurs ne sont pas les meilleurs testeurs!
 - Exception : les tests unitaires
- Les défauts sont générés par le processus de développement dans son ensemble et non pas par un individu précis (développeur)
- Le test est utilisé pour accepter ou refuser des livrables

5 - *Principes de base du test*



Indépendance
testeurs / développeurs



Tester au plus tôt et au plus
près des causes d'erreur



Harmonie avec le
développement



Résultats prouvés et
reproductibles



Automatisation

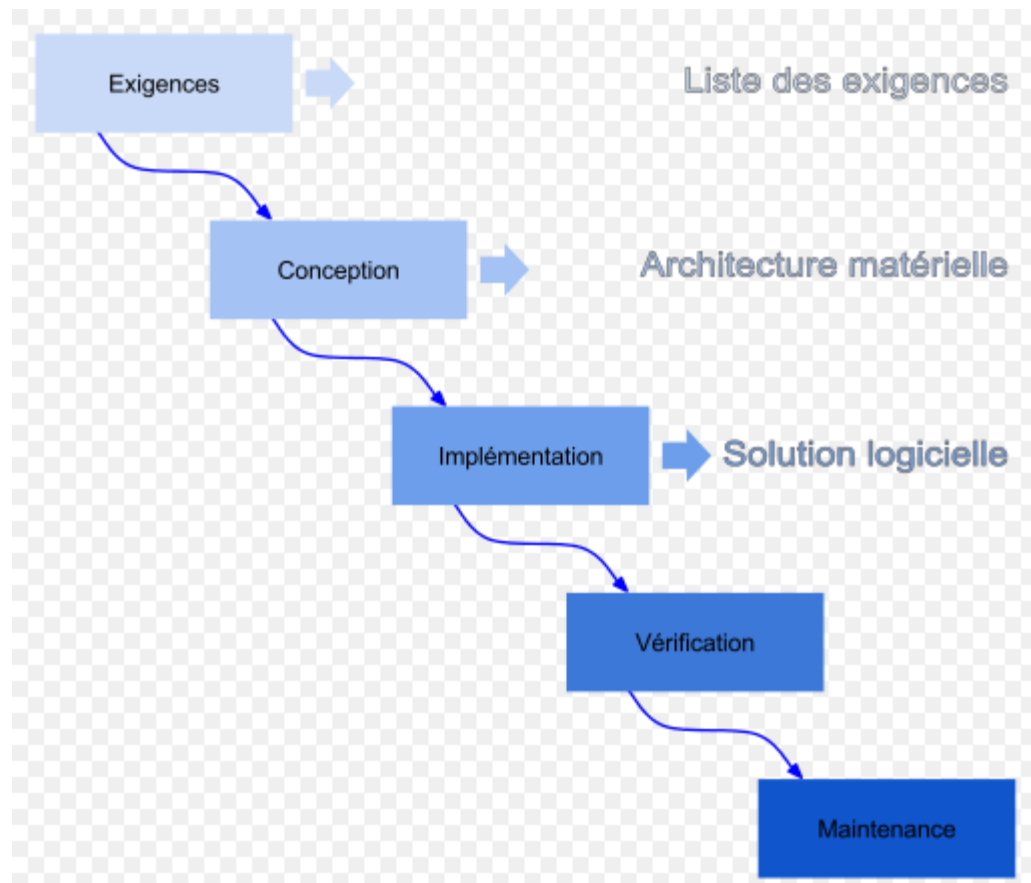
6 - *Test et développement*

- Le test est une activité structurée comme :
 - un sous-projet du projet développement
 - un projet distinct du projet développement (mais synchronisé avec lui) de façon quasi indissociable du développement
- Le test est lié au processus de développement

6 - Test et développement (1)

Modèles séquentiels cascade

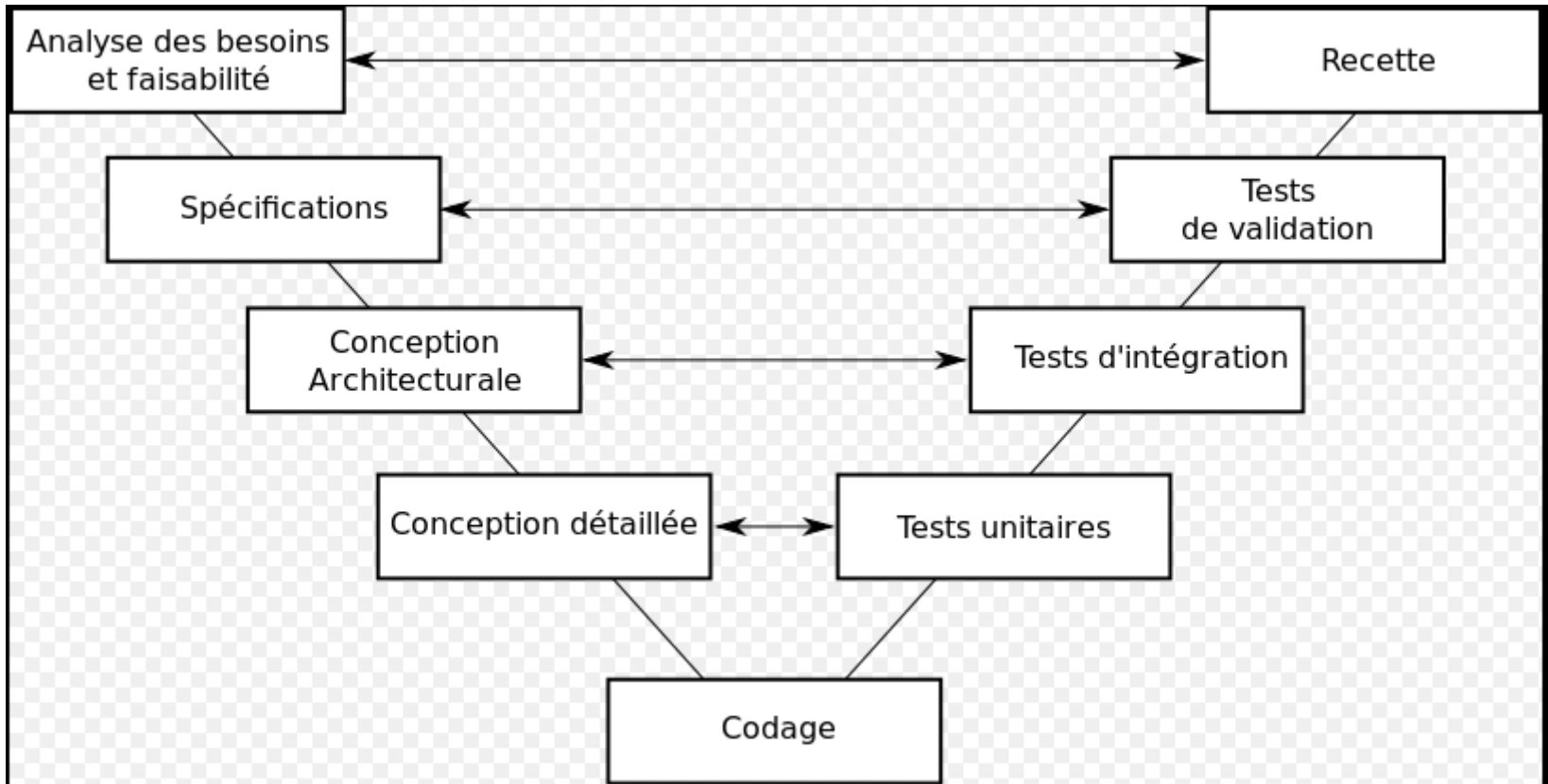
Le test est une phase spécifique



6 - Test et développement (2)

Cycle en V

- Chaque phase de réalisation est associée à un niveau de test





6 - Test et développement (3)

- **Modèles itératifs**

- ☐ Chaque itération contient une activité de test

- **Modèles incrémentaux**

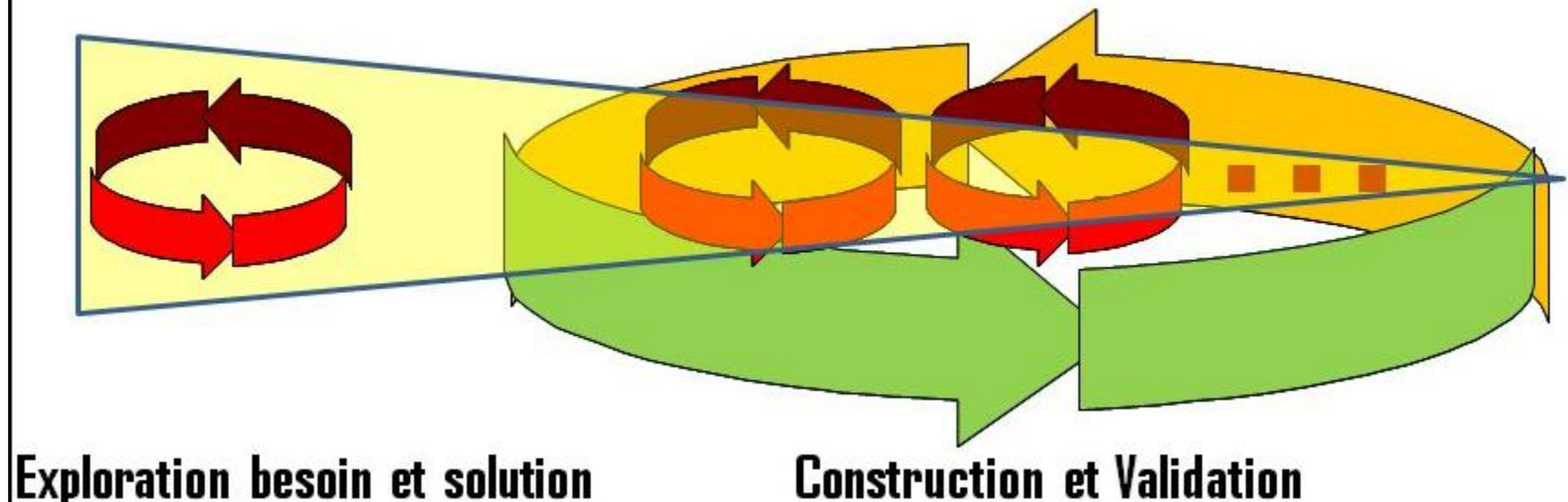
- ☐ Tous les incréments successifs sont testés

6 - Test et développement (4)

Modèles agiles

- Le test continue est un moyen nécessaire pour atteindre l'agilité
 - Certaines méthodes, le test est le moteur du développement (TDD)

L'aboutissement : un cycle adopté par l'ensemble des méthodes Agiles actuelles



7 - *Objectif des tests*

- L'objectif du test est de valider le bon fonctionnement d'un logiciel par rapport aux besoins et aux exigences recueillis auprès des utilisateurs.
- Il permet d'assurer un niveau de qualité suffisant pendant le cycle de développement et de vie d'un logiciel.
- La phase de test permet notamment d'identifier des dysfonctionnements, des anomalies ou des régressions qui empêchent la couverture totale ou partielle des exigences formulées initialement et au cours des évolutions fonctionnelles.
- L'outil de test logiciel intervient à ce moment afin de structurer, d'organiser et de cibler les tests (risques, priorités...etc).

7 - Objectif des tests

Economiser le temps
et l'argent en
identifiant
rapidement les
défaillances

Rendre le
développement plus
efficace

Augmenter la
satisfaction du client

Correspondre le
résultats aux attentes

Identifier les
modifications à
inclure dans les
prochaines versions

Identifier les
composants et les
modules réutilisables

Identifier les lacunes
des développeurs

8 - Le plan de test

- Le plan de test définit la stratégie et la portée des tests.
- Il détermine le déroulement des tests durant le projet.
- C'est le document d'entrée indispensable pour savoir :
 - ☐ ce qui est à tester,
 - ☐ par qui,
 - ☐ comment,
 - ☐ quand et jusqu'à quand.

8- 1- Objectif d'un plan de test

- Le plan de test recense les objectifs et les moyens pour réaliser les tests.
- Il permet l'organisation technique des tests,
- il planifie leur déroulement dans le temps et définit les points de repère, en particulier les conditions d'arrêt.
- Il sert aussi de document de validation de la qualité finale du logiciel et fait partie des documents contractuels du projet, au même titre que les spécifications techniques ou les besoins fonctionnels.
- Il est conçu par le responsable des tests et validé par le chef de projet.

8- 1 - Objectif d'un plan de test

- Le plan de test doit lister précisément quels objectifs sont à atteindre. Par exemple :
 - Trouver le maximum d'anomalies,
 - Identifier les problèmes importants et évaluer les risques associés,
 - Communiquer sur la qualité d'un produit et la satisfaction client,
 - Communiquer sur les tests,
 - Définir les étapes du processus de test,
 - Valider un niveau de disponibilité et la robustesse attendu du produit,
 - Valider l'adéquation du produit avec le SLA attendu, et/ou le niveau de qualité perçu par les utilisateurs.

8- 2- Les pré requis

- Les différents pré requis à la mise en œuvre d'un plan de test sont les suivants :
 - Phase d'initialisation terminée (réunion de démarrage) et donc prise de connaissance du contexte, des besoins et des contraintes,
 - Analyse des exigences et des spécifications logicielles,
 - Interviews focalisés (Chef de projet, Responsable qualité, Utilisateurs, ...).



8- 3- Introduction

- L'introduction présente les informations générales du plan :
 - Présentation de l'application
 - Contexte de réalisation
 - Choix technologiques
 - Coordonnées du responsable des tests
-
- Réalisation d'un glossaire, dans le but d'avoir un langage commun tout au long de l'activité de test,

8- 4- Projet à tester

- Il s'agit de préciser le projet
 - Liste des modules et/ou des lots à tester
 - Date de livraison



8- 5- Environnement de test

- Sites de test
- Configurations matérielles
- Outils de test
- Bases de test
- Données de test

8- 6- Tests à réaliser

- Listes des modules à tester
- Objectif des tests
- Exigences
- Analyse du risque
- Matrice Exigences/Risques pour définir les priorités
- Estimation de la charge



8- 7- Stratégie de tests

la démarche mise en œuvre pour réaliser les tests

- Description de l'approche générale
- Phase de tests
- Campagne de test
- Ordre d'exécution des tests



8- 8- Gestion des fiches d'anomalies

- Actions et états
- Gestion des flux
- Liste des intervenants

8- 9- La gestion des Ressources

- Le plan de test doit identifier précisément les ressources matérielles et humaines nécessaires pour atteindre l'objectif initial de l'activité de test. Pour les **ressources matérielles**, il faudra par exemple lister :
 - Les configurations matérielles (Serveurs, PC, banc de test, ...),
 - Les configurations logicielles (logiciels, versions, ...),
 - Les outils de production et de support (Bug tracking, Test Management, Automate de Functional Testing,...),
 - Les pré requis fonctionnels et techniques (paramétrage initial, jeux de données en entrée, ...),
 - Les équipes en jeux et les différentes responsabilités dans le processus (identification, dimensionnement et interactions)

8- 10- Le périmètre d'intervention

- Le plan de test doit définir de manière claire le périmètre d'intervention de l'activité de recette associé au projet.
C'est le QUOI de la stratégie de test.
Devront être identifiés en fonction des besoins :
 - Les éléments sur lesquels vont porter l'activité de test,
 - Les éléments qui sont exclus de la stratégie
 - Les briques fonctionnelles et techniques (flexibilité / changement et modularité)

8- 11- La définition des livrables

- Cette étape permet d'identifier les différents documents produits lors des phases de test :
 - Le cahier de recette
 - Le journal de test : Résultats intermédiaires de campagne à intervalle régulier
 - Le reporting de la couverture de test
 - Le reporting de la qualité perçue
 - Les anomalies

8- 12- Exemple de plan de tests

Élément	Description	Objectif
Responsabilités	Acteurs et affectation	Décrit qui fait quoi dans les tests. Assure le suivi et les affectations
Tests	Portée de tests, plannings, durées et priorités	Définit le processus et détaille les actions à entreprendre
Communication	Plan de communication	Tout le monde doit savoir ce qu'il doit savoir avant les tests et ce qu'il doit faire savoir après les tests
Analyse de risques	Éléments critiques à tester	Identification des domaines qui sont critiques dans le projet
Traçage des défaillance	Documentation des défaillances	Documenter les défaillances et les détails les concernant

9 - L'équipe de testeurs

- Les tests ne sont pas le travail d'un seul homme mais celui d'une équipe
- La taille de cette équipe dépend de la taille et de la complexité du projet
- Les développeurs doivent avoir un rôle dans les tests mais d'une façon réduite
- Le testeur doit être minutieux, critique(pas au sens jugement), curieux doté d'une bonne communication, rigoureux

9 - L'équipe de testeurs

- Les testeurs doivent poser des questions que les développeurs peuvent trouver embarrassantes

Comment
pouvez-vous dire
que ça marche ?

Que veut dire
pour vous « ça
marche » ?

Pourquoi ça
marchait et que
ça ne marche
plus ?

Qu'est-ce qui a
causé le mauvais
fonctionnement ?



9 - L'équipe de testeurs

- Le **coordinateur de tests** crée les plans de tests et les spécifications de tests sur la base des spécifications techniques et fonctionnelles
- Les **testeurs** exécutent les tests et documentent les résultats



10 - Ecriture des tests

- Cas de test
- Utilisation d'outil d'écriture de tests

10 – 1 Cas de test

- Un cas de test est un ensemble *d'entrées de tests*, de *conditions d'exécution* et de *résultats attendus* pour un objectif particulier tel que la conformité du programme avec une spécification données

10- 2 Anatomie d'un cas de test

<i>ID</i>
<i>Description</i>
<i>Priorité</i>
<i>Préconditions</i>
<i>Scénario</i>
<i>Résultats attendus</i>
<i>Résultats actuels</i>
<i>Remarques</i>

10- 2 Anatomie d'un cas de test

- L'ID est un *numéro unique* qui permet la traçabilité des cas de test, les lier à des spécifications ou à d'autres cas de test
- Les préconditions déterminent les *conditions nécessaires* à un cas de test. Elles indiquent aussi les *cas de test qui doivent être exécutés précédemment*
- La description est un texte *décrivant* le tests et ses attentes
- Le scénario détermine les *étapes détaillées* à suivre par le testeur
- Les résultats attendus sont *attendus* de l'exécution du scénario
- Les résultats actuels sont les *vrais résultats* obtenus, s'ils sont différents des résultats attendus, une défaillance est signalée
- Les remarques sont signalées par le testeur pour ajouter des information concernant une exécution donnée

10- 2 Anatomie d'un cas de test

- Le scénario est composé de plusieurs *actions numérotées*
- Chaque action *peut* avoir un *résultat attendu*
- L'exécution de test affecte un *résultat actuel* à chaque étape attendant un résultat
- Si chaque résultat actuel est *conforme* au résultat attendu alors le test *réussit* sinon le test *échoue*

10- 2 Anatomie d'un cas de test

Exemples

ID	Description	Préconditions	Scénario	Résultats attendus	Résultats actuels	Remarques
1	Ouverture d'un document	Existence d'un document	1- Cliquez sur le bouton « Ouvrir » 2- Sélectionnez le fichier désiré 3- Appuyez sur OK	2- Une boîte d'ouverture de fichiers s'affiche 3- Le Document est ouvert et prêt à être manipulé		
2	Impression d'un document	TC 1	1- Cliquez sur le bouton imprimer 2- Sélection la config puis cliquer sur « Imprimer »	1- La boîte de configuration d'imprimante s'affiche 2- Le document sort sous format papier		



TestLink



10- 3 TestLink

- Un outil de gestion de cas de tests (Test Case Manager).
- Une plateforme web (Open Source) écrite en PHP, qui vous permet d'organiser vos cas de tests sous forme de plans de tests.
- Il permet de centraliser toute la gestion des tests fonctionnels autour d'une unique interface web accessible à tout moment, et à toute l'équipe de projet (clients, prestataires, etc..).

11 – 1- Défaut (bug)

- Un *défaut* est une imperfection dans un composant ou un système qui peut en perturber le fonctionnement selon L'ISTQB(*Standard glossary of terms used in Software Testing*)
- Un code peut être syntaxiquement et sémantiquement correct mais présente un défaut qui ne sera manifesté que lors d'un test de performance par exemple => une erreur d'architecture ou de configuration.



11 - 2 - Priorisation des défaillances

- Le plan de tests détermine la **priorisation** des défaillances
- La priorisation facilite la communication entre développeurs et testeurs ainsi que l'affectation et la planification des tâches

11 - 2 - Priorisation des défaillances

Priorité	Description
1 - Fatal	Impossible de continuer les tests à cause de la sévérité des défaillances
2- Critique	Les tests peuvent continuer mais l'application ne peut passer en mode production
3- Majeur	L'application peut passer en mode production mais des exigences fonctionnelles importantes ne sont pas satisfaites
4- Medium	L'application peut passer en mode production mais des exigences fonctionnelles sans très grand impact ne sont pas satisfaites

11 - 2 - Priorisation des défaillances

Priorité	Description
5- Mineur	L'application peut passer en mode production, la défaillance doit être corrigée mais elle est sans impact sur les exigences métier
6- Cosmétique	Défaillances mineures relatives à l'interface (couleurs, police, ...) mais n'ayant pas une relation avec les exigences du client

12 Types de tests

Tests unitaires

Tests
d'intégration

Tests du
système et de
fonctionnement

Tests
d'acceptation

Tests de
régression

Béta-Tests

12 *Les types de tests*

- Définition du test

- ☐ Processus manuel ou automatique, vérifiant l'adéquation d'un système vis-à-vis de sa spécification

- Les tests mettent en évidence les bugs

- ☐ Sans en diagnostiquer les causes
- ☐ Sans les corriger



12 – 1 Les tests unitaires

- Les tests unitaires consistent à tester individuellement les composants de l'application (classes ou méthodes) à fin de valider la qualité du code et les performances d'un module.



12 – 1 Tests unitaires

- Les tests unitaires sont les tests de blocs individuels (par exemple des blocs de code)
- Les tests unitaires sont généralement écrits par les développeurs eux-mêmes pour la validation de leurs classes et leurs méthodes
- Les tests unitaires sont exécutés généralement par les machines

12 – 1 Tests unitaires/tests de recettes

■ Tests unitaires

- ☐ Spécifiés par l'équipe
- ☐ Garantissent le bon fonctionnement et permettent l'intégration continue

■ Tests de recette

- ☐ Spécifiés par le client
- ☐ Permettent au client et au(x) manager(s) de mesurer l'avancement du projet
- ☐ Supposent une exécution automatique
 - intégration de nouveaux tests quasi permanente
 - Nombre important de tests contenus dans un projet

12 – 1 Tests unitaires/tests de recettes

- Tests unitaires

- ☐ Facilitent le développement incrémental
- ☐ Permettent l'amélioration permanente

- Tests de recettes

- ☐ Renforce l'efficacité du développement et la bonne évaluation de l'avancement

12 – 2 La revue de code

- Une revue de code peut s'appuyer sur la vérification (manuelle ou automatisée) **du respect d'un ensemble de règles de programmation.**
- Elle doit permettre une relecture du code de l'application (souvent par des outils automatiques qui inspectent le code par rapport à des normes prédéfinies)
-

12 – 2-a Objectif de revue de code

- améliorer la qualité du code
- favoriser la collaboration, le travail en équipe('appropriation du code par l'équipe)
- appliquer un standard
- détecter et corriger les défauts (bugs mais aussi lisibilité) au plus tôt dans le cycle de vie du code pour économiser les coûts
- formation des développeurs

12 – 2-b Utilisation d'un outil dédié

- Il existe des logiciels permettant d'assister la revue de code:
 - collecte et présentation des modifications apportées aux fichiers sources qui nécessitent une relecture
 - gestion du workflow : la relecture devient une étape de l'intégration continue
 - annotation des défauts et commentaires issus de la relecture, suivi des corrections de ces défauts
 - statistiques et métriques (vitesse de relecture, taux de détection des défauts...)

12 – 2-b Utilisation d'un outil dédié

- Quelques logiciels libres:
- Review Board, logiciel libre à interface web créé par VMware.
- Agile Review, plugin libre pour Eclipse.
- Crew : logiciel libre de revue de code pour les projets Git.
- Rietveld développé pour Google par Guido van Rossum
- CodeStriker
- JCR (Java Code Reviewer)

12 – 3 Les tests d'intégration

- Ces tests sont exécutées pour valider l'intégration des différents modules entre eux et dans leur environnement d'exploitation définitif.
- Ils permettront de mettre en évidence des problèmes d'interfaces entre différents programmes.



12 – 4 Les tests d'installation

- Une fois l'application validée, il est nécessaire de contrôler les aspects liés à la documentation et à l'installation.
- Les procédures d'installation doivent être testées intégralement car elles garantissent la fiabilité de l'application dans la phase de démarrage.
- Bien sûr, il faudra aussi vérifier que les supports d'installation ne contiennent pas de virus.



12 – 5 Les tests fonctionnels

- Ces tests ont pour but de vérifier la conformité de l'application développée avec le cahier des charges initial. Ils sont donc basés sur les spécifications fonctionnelles.

12 – 6 Les tests IHM

- Les tests IHM ont pour but de vérifier que la charte graphique a été respectée tout au long du développement.

Cela consiste à contrôler :

- ☐ la présentation visuelle : les menus, les paramètres d'affichages, les propriétés des fenêtres, les barres d'icônes, la résolution des écrans, les effets de bord,...
- ☐ la navigation : les moyens de navigations, les raccourcis, le résultat d'un déplacement dans un écran,...

12 – 7 Les tests de configuration

- Une application doit pouvoir s'adapter au renouvellement de plus en plus fréquent des ordinateurs. Il s'avère donc indispensable d'étudier l'impact des environnements d'exploitation sur son fonctionnement.

Voici quelques sources de problèmes qui peuvent surgir lorsque l'on migre une application vers un environnement différent :

- l'application développée en 16 bits migre sur un environnement 32 bits,
- les DLL sont incompatibles,
- les formats de fichiers sont différents,
- les drivers de périphériques changent,
- les interfaces ne sont pas gérées de la même manière...

12 – 8 Les tests de performance

- Le but principal des tests de performance est de valider la capacité qu'ont les logiciels, les serveurs et les réseaux à supporter des charges d'accès importantes.
- On doit notamment vérifier que les temps de réponse restent raisonnable lorsqu'un nombre important d'utilisateurs sont simultanément connectés à la base de données de l'application.
- Pour cela, il faut d'abord relever les temps de réponse en utilisation normale, puis les comparer aux résultats obtenus dans des conditions extrêmes d' utilisation.
- Nombre d'utilisateurs connectés / actifs
- Volume de données en base



12 – 8 Les tests de performance

- Une solution est de simuler un nombre important d'utilisateur en exécutant l'application à partir d'un même poste et en analysant le trafic généré.
- Le deuxième objectif de ces tests est de valider le comportement de l'application, toujours dans des conditions extrêmes. Ces tests doivent permettre de définir un environnement matériel minimum pour que l'application fonctionne correctement.

12 – 9 Test d'endurance

- Vérifier le comportement du système dans le temps
 - L'absence de fuite mémoire
 - Pas de dégradation de comportement (temps d'accès , temps de traitement,
 - La place mémoire (base de données et système
 - Les performances au cours du temps

12 – 10 Test de charge

- Vérifier que le système se comporte comme prévu
- Performance,
- Service
- Dans les conditions de charge admise

12 – 11 Test de stress

- Vérifier le comportement du système dans des conditions hors limites définis:
- Charges anormales
- Pic
- Erreurs élevés
- Ces tests permettent de trouver des anomalies difficiles à trouver et vérifie les critères de type fiabilité



12 – 12 Les tests de non-régression

- Les tests de non-régression permettent de vérifier que des modifications n'ont pas altérées le fonctionnement de l'application.
- L'utilisation d'outils de tests, dans ce domaine, permet de faciliter la mise en place de ce type de tests.

Comparaison entre les tests

Test	Portée	Exécutant
Unitaires	Petites portions du code source	Développeur Machine
Intégration	Classes / Composants	Développeur
Fonctionnel	Produit	Testeur
Système	Produit / Environnement simulé	Testeur
Acceptation	Produit / Environnement réel	Client
Beta	Produit / Environnement réel	Client



Ecriture d'un test unitaire

1- Cas de tests unitaire

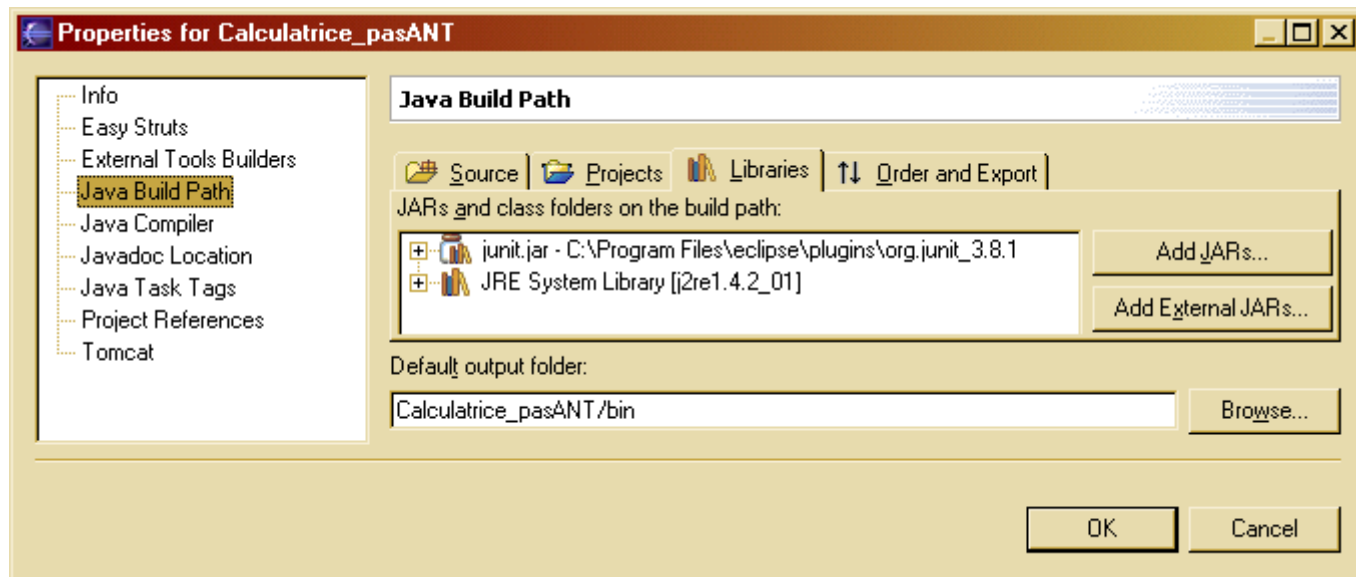
- Au moins 3 catégories de cas à tester :
 - Les Cas nominaux / en succès
 - Vérifier que la méthode fait le job en fonctionnement normal
 - Les Cas d'erreurs
 - Vérifier que la méthode gère bien les erreurs
 - Les cas au limites / tordus / peu communs
 - Vérifier que la méthode est robuste

2- Génération ou création de la classe du test unitaire

- La technique pour créer un test consiste à :
 - créer une nouvelle classe, portant pour des raisons de commodité le **nom de la classe** que vous voulez tester, suivi de "**Test**".
 - Cette nouvelle classe doit **hériter de la classe "TestCase"**.
- **public class** CalculatorTest **extends** TestCase {}
- Avec Eclipse, vous ne pourrez hériter de TestCase qu'en ayant ajouté le jar de JUnit à votre build path .

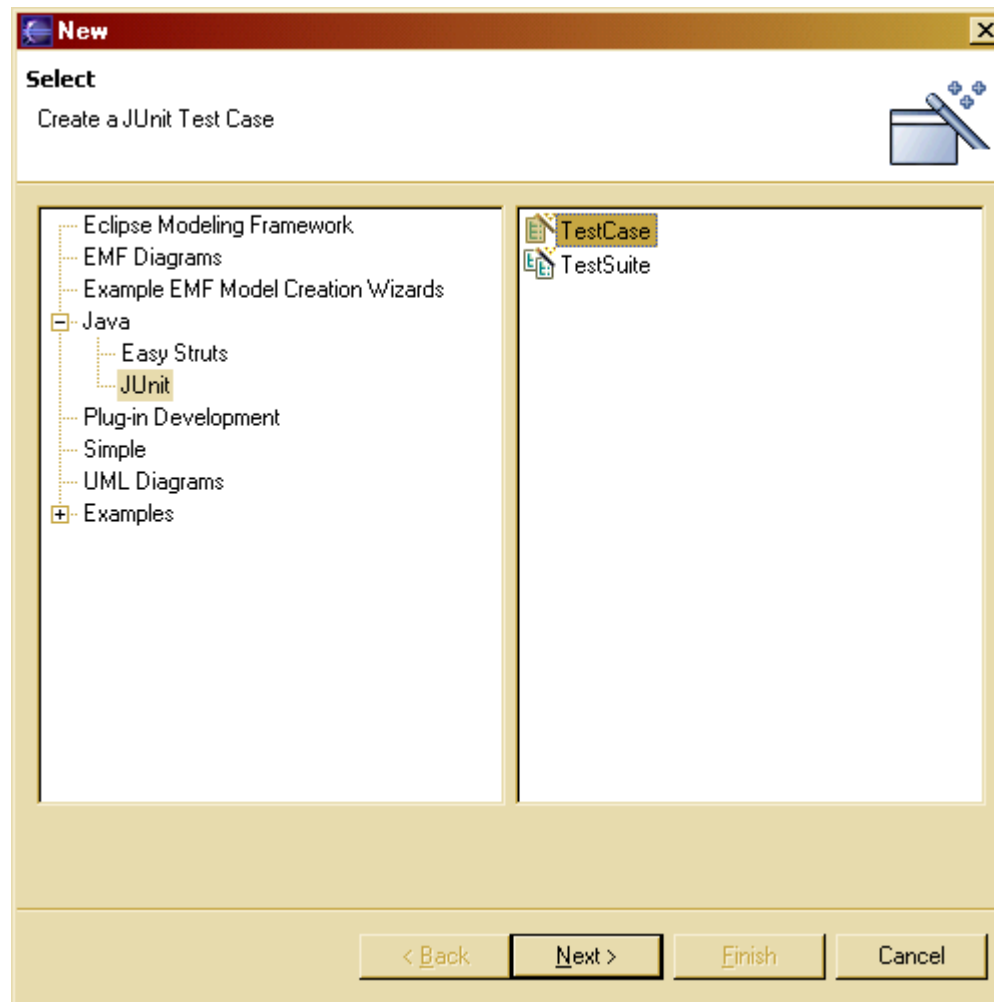
3-

Ajout jar de junit



Etape 1 :

Dans un premier temps, on ne crée que des TestCase. Les TestSuite permettent seulement de rassembler des TestCase et d'autres TestSuite.



Etape 2 :

Pensez à séparer les sources des tests des autres sources, afin de faciliter la création d'un .jar pour votre client qui ne contienne pas les tests unitaires

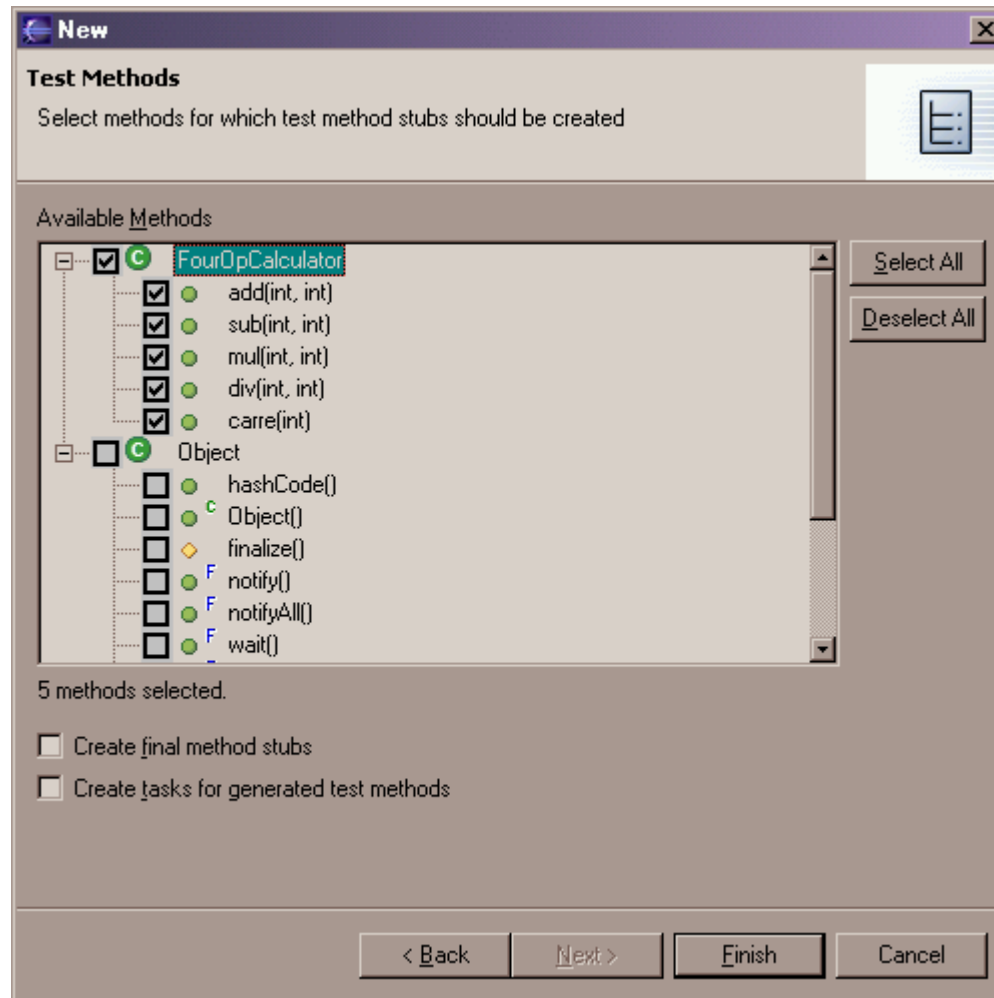
The screenshot shows a 'New' dialog box with the title 'JUnit TestCase'. The subtitle is 'Create a new JUnit TestCase'. The dialog is divided into several sections:

- Source Folder:** A text field containing 'Calculatrice_pasANT/src/main' and a 'Browse...' button.
- Package:** A text field containing 'fr.umlv.exposeJUnit.calculators' and a 'Browse...' button.
- Test case:** A text field containing 'FourOpCalculatorTest'.
- Test class:** A text field containing 'fr.umlv.exposeJUnit.calculators.FourOpCalculator' and a 'Browse...' button.
- Superclass:** A text field containing 'junit.framework.TestCase' and a 'Browse...' button.
- Which method stubs would you like to create?**
 - ☐ public static void main(String[] args)
 - ☐ Add TestRunner statement for: text ui
 - ☐ setUp()
 - ☐ tearDown()

At the bottom of the dialog, there are four buttons: '< Back', 'Next >', 'Finish', and 'Cancel'.


Etape 3, facultative :

Les racines des méthodes à tester peuvent être générées automatiquement



3. Ecriture du test unitaire

- **La méthode** doit avoir un nom débutant par "test".
- Elle doit être déclarée **public**, et ne rien renvoyer (**void**).
- Eclipse permet de générer automatiquement ces méthodes



```
public class Calculator {  
  public int add(int a, int b)  
  {  
    return a + b;  
  }  
  public int sub(int a, int b)  
  {  
    return a - b;  
  }  
  public int mul(int a, int b)  
  {  
    return a * b;  
  }  
  public int div(int a, int b)  
  {  
    return a / b;  
  }  
}
```

Exemple d'une méthode de test

```
public void testSub() {  
    int result = a - b;  
    assertEquals(result, simpleCalculator.sub(a,  
        b));  
}
```

Dans `assertEquals`, généralement on met la **valeur de référence attendue** par le test en premier argument, et en second argument, **la valeur obtenue auprès de la méthode testée**.

Message d'erreur généré quand le test ne passe pas

- Ce test "passe" si les deux arguments de assertEquals sont égaux .
- Dans le cas contraire, un message d'erreur est généré comme suit:

```
junit.framework.AssertionFailedError: expected:<8> but was:<7>  
at junit.framework.Assert.fail(Assert.java:47)  
at junit.framework.Assert.failNotEquals(Assert.java:282)  
at junit.framework.Assert.assertEquals(Assert.java:64)  
at junit.framework.Assert.assertEquals(Assert.java:201)  
at junit.framework.Assert.assertEquals(Assert.java:207)  
at fr.umlv.exposeJUnit.calculators.FourOpCalculatorTest.testAdd(FourOpCalculatorTest.java:45)
```



4. Lancement du test unitaire

- Comment savoir si le test qu'on vient d'écrire passe ?

4.1 Eclipse

- Avec l'IDE Eclipse, la vue graphique s'obtient facilement: Il suffit de lancer le TestCase en temps que "JUnit Test" (Run As... JUnit Test).
- Par défaut, une barre rouge est affichée en cas de problème, et un message dans la barre d'état en cas de réussite.

4.2 En ligne de commande

- Sans l'aide d'Eclipse, vous devez compiler le test. Le test ne s'exécute pas tel quel : il n'y a pas de méthode main.

```
C:\workspace_junit\calculatrice_pasAnt\bin>java fr\umlv\exposeJUnit\calculators\FourOpCalculatorTest
Exception in thread "main" java.lang.NoClassDefFoundError:
fr\umlv\exposeJUnit\calculators\FourOpCalculatorTest
(wrong name: fr/umlv/exposeJUnit/calculators/FourOpCalculatorTest)
```

- Il faut lancer en fait un "TestRunner", qui se trouve dans le .jar de JUnit.
- Il existe plusieurs TestRunner, un pour chaque type d'interface. Vous avez le choix entre une interface en mode texte, deux en mode graphique (awt ou swing).

4.2.1 Interface d'affichage texte (texteui)

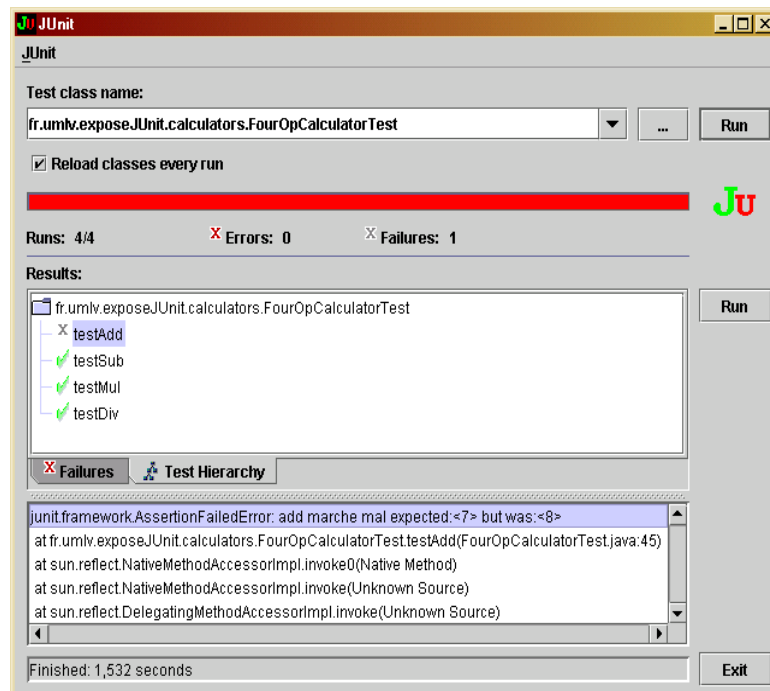
```
C:\workspace_junit\calculatrice_pasAnt\bin>
java -cp .;c:\junit\junit3.8.1\junit.jar junit.textui.TestRunner f

r.uml.v.exposeJUnit.calculators.FourOpCalculatorTest
.F...
Time: 0,02
There was 1 failure:
1) testAdd(fr.uml.v.exposeJUnit.calculators.FourOpCalculatorTest) junit.framework.
AssertionFailedError: add marche mal expected:<7> but was:<8>
    at fr.uml.v.exposeJUnit.calculators.FourOpCalculatorTest.testAdd(FourOpCa
lculatorTest.java:45)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)

FAILURES!!!
Tests run: 4,  Failures: 1,  Errors: 0
```

4.2.2 Interface d'affichage graphique (swingui) :

- Le TestRunner en swing permet de charger des TestCase et des TestSuite. Il affiche sous la forme d'un arbre le résultat du test unitaire
- En paramètre du TestRunner, mettez le nom de la classe de test.



5.1 La classe Assert

Le propre d'un test unitaire est d'échouer quand le code testé ne fait pas ce qui est prévu. Pour faire échouer un test JUnit (une des méthodes testXXX d'un TestCase), il faut utiliser une des méthodes de la classe junit.framework.Assert, qui sont toutes accessibles au sein d'un TestCase.

- **5.1.1 assertEquals** : Permet de tester si deux types primitifs sont égaux (boolean, byte, char, double, float, int, long, short).
- **5.1.2 assertFalse et assertTrue** : Teste une condition booléenne.
- **5.1.3 assertNotNull et assertNull** : Teste si une référence est non nulle.
- **5.1.4 assertNotSame et assertEquals** : Teste si deux Object se réfèrent ou non au même objet.
- **5.1.5 fail**: Fait échouer le test sans condition. Il est conseillé de mettre un message expliquant pourquoi le test a échoué

5.2 Personnalisation des messages d'erreur

- Les messages d'erreur peuvent être personnalisés. Les méthodes de test existent toutes sous deux formes :
 - l'une qui ne prend pas de message,
 - et l'autre qui en prend au niveau de son premier paramètre. On obtient un message de ce type :
 - junit.framework.AssertionFailedError: add marche mal expected:<7> but was:<8> at junit.framework.Assert.fail(Assert.java:47) at junit.framework.Assert.failNotEquals(Assert.java:282) at junit.framework.Assert.assertEquals(Assert.java:64) at junit.framework.Assert.assertEquals(Assert.java:201) at fr.umlv.exposeJUnit.calculators.FourOpCalculatorTest.testAdd(CalculatorTest.java:45)

5.3 Fixture : la mise en place des tests avec setUp et tearDown

- Une grande partie du code d'un test unitaire sert à établir les conditions d'exécution du test.
- Au sein d'un même TestCase, il peut arriver que toutes les méthodes de test aient besoin d'un minimum de chose (une connexion à une base de données par exemple).
- La mise en place de ces conditions est prévue par le framework JUnit. Plutôt que chacun de vos tests appelle une méthode de mise en place, puis une méthode de nettoyage, le framework JUnit lance automatiquement avant un test la méthode setUp et après le test la méthode tearDown.

5.3 Fixture : la mise en place des tests avec setUp et tearDown

- Prenons l'exemple d'une classe gérant la copie de fichier.
- Tous les tests vont avoir besoin :
 - d'un jeu de fichiers de test,
 - L'effacés à l'issu de chaque test.
- Il est intéressant dans ce cas d'implémenter dans le TestCase les méthodes setUp et tearDown, qui vont respectivement créer le jeu de fichiers de test et le détruire.

```
* Test case de l'opération de copie
*/
```

```
public class CopyOperationTest extends TestCase {
```

```
    protected String[] filesNames = { "test0", "test1", "test2", "test3", "test4", "test5" };
    protected File testFolder;
```

```
    /**
```

```
     * Constructor for CopyOperationTest.
```

```
     * @param arg0
```

```
     */
```

```
    public CopyOperationTest(String arg0) {
        super(arg0);
```

```
    }
```

```
    /**
```

```
     * @see TestCase#setUp()
```

```
     */
```

```
    protected void setUp() throws Exception {
        super.setUp();
```

```
        testFolder = new File("argonaute_test_CopyOperation");
```

```
        testFolder.mkdir();
```

```
        for (int i = 0; i < filesNames.length; i++) {
```

```
            File newFile = new File(testFolder.getPath() + "/" + filesNames[i]);
```

```
            newFile.createNewFile();
```

```
        }
```

```
    }
```

```
    /**
```

```
     * @see TestCase#tearDown()
```

```
     */
```

```
    protected void tearDown() throws Exception {
```

```
        super.tearDown();
```

```
        for (int i = 0; i < filesNames.length; i++) {
```

```
            File newFile = new File(testFolder.getPath() + "/" + filesNames[i]);
```

```
            newFile.delete();
```

```
        }
```

```
        testFolder.delete();
```

```
    }
```

```
    ...
```

```
}
```

6 Compilation et tests unitaires dans une même opération avec les TestSuites et Ant

- Le test permet de savoir si le code fait bien ce qu'on lui demande. La compilation permet de savoir si la syntaxe du code est correcte. Pourquoi ne pas compiler et tester dans une même opération ? C'est ce qu'il est possible de faire avec Ant et un TestSuite qui va nous permettre d'agréger tous les tests créés.
- **6.1 TestSuite**
- Réaliser un groupement de tests est simple. La classe dite "TestSuite" n'hérite pas en fait de TestSuit. Elle définit une méthode publique, nommée "suite", renvoyant un objet de type Test qui peut contenir un TestCase ou un TestSuite.
 - Attention, pour réaliser vos "TestSuite", vous devez reprendre le même prototype de fonction.
 - Eclipse propose un Wizard automatisant la création du TestSuite.

Exemple de test suite

Créer une classe java : MessageUtil.java in C:\ > **JUNIT_WORKSPACE**

```
/*
 * This class prints the given message on console.
 */
public class MessageUtil {

    private String message;

    //Constructor
    //@param message to be printed
    public MessageUtil(String message) {
        this.message = message;
    }

    // prints the message
    public String printMessage() {
        System.out.println(message);
        return message;
    }

    // add "Hi!" to the message
    public String salutationMessage() {
        message = "Hi!" + message;
        System.out.println(message);
        return message;
    }
}
```

Créer la classe TestJunit1.java sur C:\ > JUNIT_WORKSPACE

```
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

public class TestJunit1 {

    String message = "Robert";
    MessageUtil messageUtil = new MessageUtil(message);

    @Test
    public void testPrintMessage() {
        System.out.println("Inside testPrintMessage()");
        assertEquals(message, messageUtil.printMessage());
    }
}
```


Créer la classe TestJunit2.java sur C:\ > JUNIT_WORKSPACE

```
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

public class TestJunit2 {

    String message = "Robert";
    MessageUtil messageUtil = new MessageUtil(message);

    @Test
    public void testSalutationMessage() {
        System.out.println("Inside testSalutationMessage()");
        message = "Hi!" + "Robert";
        assertEquals(message, messageUtil.salutationMessage());
    }
}
```

Créer la classe Test Suite : TestSuite.java

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestJUnit1.class,
    TestJUnit2.class
})
public class JunitTestSuite {
}
```

Créer la classe Test Runner : TestRunner.java

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(JunitTestSuite.class);
        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }
        System.out.println(result.wasSuccessful());
    }
}
```

Compiler toutes les classes avec javac

```
C:\JUNIT_WORKSPACE>javac MessageUtil.java TestJunit1.java  
TestJunit2.java JunitTestSuite.java TestRunner.java
```

Exécuter le Test runner

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Résultat de test

```
Inside testPrintMessage()  
Robert  
Inside testSalutationMessage()  
Hi Robert  
true
```