Documentation informatique

MÉTHODOLOGIES DE DÉVELOPPEMENT

LES TESTS UNITAIRES EN GENIE LOGICIEL

VERSION	OBJET	DATE
1.0	Création	30 Juillet 2018

Table des matières

I. INTRODUCTION:....4
I.1. IMPORTANCE DES TESTS UNITAIRES:....4
I.2. PLAN DE L'OUVRAGE....4
II. GÉNÉRALITÉS:....5
II.1. NOTION DE TEST UNITAIRE:....5
II.1.1. DÉFINITIONS:....5
II.1.2. DOMAINES D'EMPLOI:....6
II.2. SIMILITUDES ET DIFFÉRENCES AVEC LES TESTS DE RÉCEPTION:....7
II.2.1. DIFFÉRENCES:....7
II.2.2. SIMILITUDES:....8
III. CONCEPTION ET RÉALISATION DE TESTS UNITAIRES:....9
III.1. ACTIVITÉS COMPRISES DANS LES PROCÉDURES DE TESTS UNITAIRES:....9
III.1.1. PRINCIPE:....9
III.1.2. LISTE DES ACTIVITÉS:....9
III.1.3. OBJECTIFS D'UN TEST UNITAIRE:....11
III.1.4. SIMULATION DE L'ENVIRONNEMENT D'EXÉCUTION:....12
III.1.5. CONFIGURATION DE TEST DU COMPOSANT CALCUL MENSUALITÉ:....16
III.1.6. SCHÉMA DE PRINCIPE:....18
III.1.7. RECOMMANDATIONS DIVERSES:....19
IV. ÉVALUATION DE LA COUVERTURE D'UN TEST:....20
IV.1. COUVERTURE D'UN TEST AU SENS GÉNÉRAL:....20
IV.2. COUVERTURE DE CODE:....20
IV.2.1. DÉFINITION:....20
IV.2.2. DIFFÉRENTS NIVEAUX DE TESTS DE COUVERTURE DE CODE:....21
IV.2.3. LA MÉTHODE MC/DC:....25
IV.2.4. OUTILS DE COUVERTURE DE CODE:....26
V. PLACE DES TESTS UNITAIRES DANS LE PROCESSUS DE DÉVELOPPEMENT:....27
V.1. QUAND FAUT-IL ÉLABORER LES TESTS?....27
V.1.1. CONTRAINTES TECHNIQUES:....27
V.1.2. ÉCRIRE LES TESTS AVANT L'OBJET A TESTER:....27
V.2. LA DÉMARCHE TEST-DRIVEN DEVELOPMENT (TDD)....27
VI. LES FRAMEWORKS DE TESTS UNITAIRES:....29
VI.1. HISTORIQUE ET GÉNÉRALITÉS:....29
VI.2. LE FRAMEWORK JUNIT:....30
VI.2.1. PRÉSENTATION:....30
VI.2.2. STRUCTURE GÉNÉRALE:....30
VI.2.3. ORGANISATION DU CODE DE TEST:....31
VI.2.4. ORGANISATION D'UNE CLASSE TESTCASE:....31
VI.2.5. LES FONCTIONS ASSERT:....32
VI.2.6. STRUCTURE DU CODE D'UNE CLASSE TESTCLASS:....33
VI.2.7. EXÉCUTION DES TESTS:....34
VI.2.8. EXEMPLE DÉTAILLÉ:....35

VI.2.9. SOLUTION POUR «FACTORISER» LES CAS DE TEST SUR DES SÉRIES DE VALEURS:.....38

I.INTRODUCTION:

I.1.IMPORTANCE DES TESTS UNITAIRES:

Parmi les activité liées au génie logiciel, les TESTS UNITAIRES sont probablement celles qui sont les moins populaires auprès des développeurs, peut être parce qu'ils ne peuvent s'empêcher de la ressentir comme une activité chronophage dont le but implicite est de remettre en cause leur travail.

Pourtant, la réalisation de tests unitaires systématiques et exhaustifs, si elle est menée en parallèle avec la réalisation des unités de logiciels qu'ils sont censé tester, permet de contrôler d'une manière très fine l'avancement des travaux de conception, c'est à dire le PLANNING de la réalisation.

Elle constitue également une condition indispensable au contrôle de la QUALITÉ du produit final d'un projet logiciel: c'est cette activité qui permet de garantir la ROBUSTESSE des logiciels produits (absence de dysfonctionnements dus à des conditions d'exécution non prévues ou à de mauvais paramétrages).

Enfin, la réalisation des tests unitaires permet souvent de mettre en évidence des erreurs, des imprécisions ou des oublis dans les documents de SPÉCIFICATIONS.

I.2.PLAN DE L'OUVRAGE

- Après une rapide présentation de l'activité de tests unitaires, cet ouvrage s'attache à décrire les principes généraux de la conception et du développement des TESTS UNITAIRES ainsi que les principaux concepts qui leur sont attachés;
- Il aborde ensuite le sujet des TESTS DE COUVERTURE, activité complémentaire des tests unitaires;
- Il s'intéresse ensuite à la pratique des tests unitaires dans les différentes phases d'un projet informatique ainsi qu'à la démarche TEST DRIVEN DEVELOPMENT (développement guidé par les tests);
- Enfin, il aborde la question des FRAMEWORKS de tests unitaires en prenant pour exemple le logiciel JUNIT, dont une rapide présentation et des exemples d'utilisation sont fournis.

II.GÉNÉRALITÉS:

II.1.NOTION DE TEST UNITAIRE:

II.1.1.DÉFINITIONS:

A. UNITÉ DE CODAGE D'UN LOGICIEL:

Pendant la phase de conception préliminaire d'un logiciel, la démarche d'ANALYSE ORGANIQUE conduit à décomposer ce logiciel en différents COMPOSANTS possédant chacun une individualité propre. Ceci veut dire que les données et traitements qu'ils encapsulent et les interfaces qu'ils offrent au reste de l'application ou à l'environnement externe sont bien identifiés. L'ensemble de ces éléments constituent les SPÉCIFICATIONS TECHNIQUE du composant considéré.

EXEMPLE:

Les flèches pleines bleues représentent des relation d'utilisation.

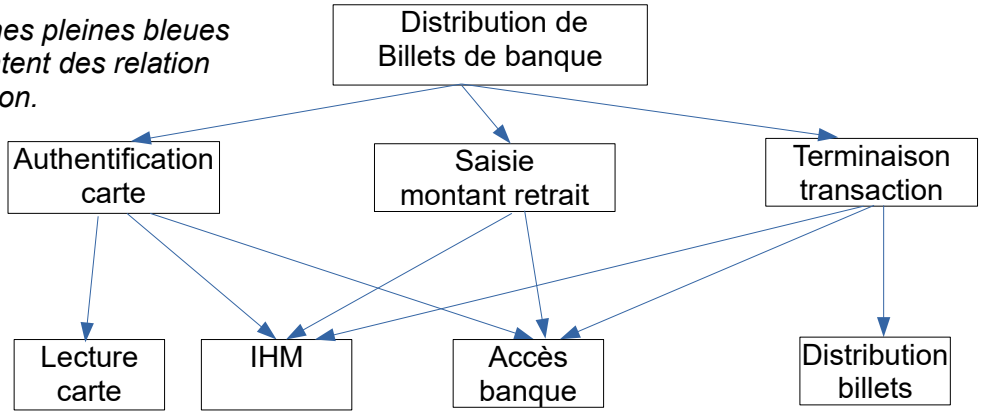


Schéma I.1.1.A

Ce type de décomposition peut être itéré pour chacun des composants trouvés au premier niveau d'analyse jusqu'à ce que les sous-composants identifiés:

- Soit correspondent à des composants existants (utilisation de composants «sur étagère» ou réutilisation de composants développés à une autre occasion);
- Soit sont suffisamment simples pour être codés directement sous forme de CLASSES ou de bibliothèques de fonctions.

B. TESTS UNITAIRES:

Dans le domaine du génie informatique, un TEST UNITAIRE est une PROCÉDURE plus ou moins automatisée qui permet de déterminer dans quelle mesure les APTITUDES d'une UNITÉ DE CODAGE d'un logiciel (c'est à dire ses SPÉCIFICATIONS TECHNIQUES) sont conformes à ce que ses concepteurs attendent dans le cadre de l'architecture du produit.

II.1.2.DOMAINES D'EMPLOI:

Les TESTS UNITAIRES sont utilisés dans toutes les méthodologies de création de logiciel, y compris lors de l'évolution d'un logiciel existant. Plus précisément:

- Ils permettent de VALIDER les COMPOSANTS (ou unités de conception) issus de la phase de CONCEPTION DÉTAILLÉE (création des différents composants du logiciels) par rapport à leurs SPÉCIFICATIONS TECHNIQUES.
- Ils sont également utilisés lors de la phase d'INTÉGRATION (phase d'agrégation des composants) pour vérifier de ces composants sont toujours valides après avoir été intégrés à l'application et que le fonctionnement des composants déjà intégrés n'est pas affecté par l'intégration des nouveaux composants (absence d'effets de bord).
- Ils permettent également de SÉCURISER les opérations de maintenance du logiciel en signalant les éventuelles RÉGRESSIONS provoquées par la modification des composants;
- Ils constituent des compléments utiles à la documentation des composants en fournissant des exemples d'utilisation de ceux-ci;
- Enfin, si les tests unitaires sont créés avant le composant à tester (ce qui est tout à fait possible car on n'a besoin pour cela que des SPÉCIFICATIONS TECHNIQUES du composant), Ils permettent de BIEN COMPRENDRE l'interface du module à développer et éventuellement de faire ressortir des erreurs de conception.

II.2.SIMILITUDES ET DIFFÉRENCES AVEC LES TESTS DE RÉCEPTION:

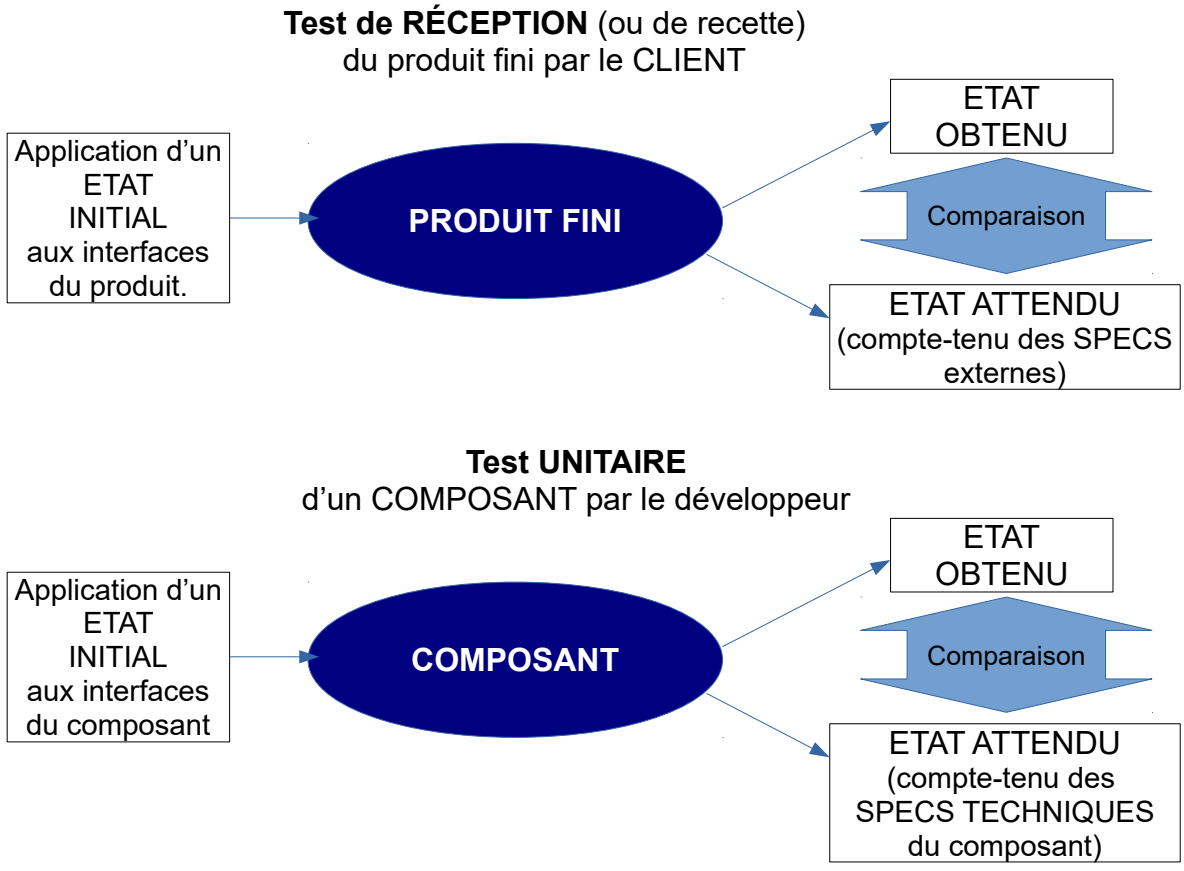
II.2.1.DIFFÉRENCES:

Nous avons vu plus haut que les TESTS UNITAIRES sont des outils qui permettent aux DÉVELOPPEURS de déterminer dans quelle mesure les APTITUDES d'une UNITÉ DE CODAGE sont conformes à ses SPÉCIFICATIONS TECHNIQUES, issues de la CONCEPTION PRÉLIMINAIRE du produit.

Les TESTS DE RÉCEPTION, permettent de vérifier que le produit fini est CONFORME aux exigences exprimées par le CLIENT (SPÉCIFICATIONS EXTERNES). Ils sont de ce fait utilisés par le CLIENT pendant la phase de RÉCEPTION (ou de recette) du produit.

RAPPEL:

LES SPÉCIFICATIONS EXTERNES donnent, comme leur nom l'indique, une vision externe du produit. Elles permettent de définir CE QUE L'ON DOIT FABRIQUER et non COMMENT ON LE FABRIQUE. Ces types de tests sont dits BOÎTE NOIRE car ils ne prennent pas en compte la structure interne du produit.



II.2.2.SIMILITUDES:

La différence entre TESTS UNITAIRES et TESTS DE RÉCEPTION se situe donc surtout dans leur domaine d'utilisation. En revanche, la conception des tests de réception obéit sensiblement aux mêmes principes que celle des tests unitaires.

III.CONCEPTION ET RÉALISATION DE TESTS UNITAIRES:

III.1.ACTIVITÉS COMPRISES DANS LES PROCÉDURES DE TESTS UNITAIRES:

III.1.1.PRINCIPE:

Un TEST UNITAIRE est une procédure qui effectue une comparaison entre le RÉSULTAT DE L'EXÉCUTION d'une fonction ou d'une méthode d'un COMPOSANT (classe ou bibliothèque de fonctions) et le RÉSULTAT qui est ATTENDU compte tenu des SPÉCIFICATIONS TECHNIQUES de ce composant.

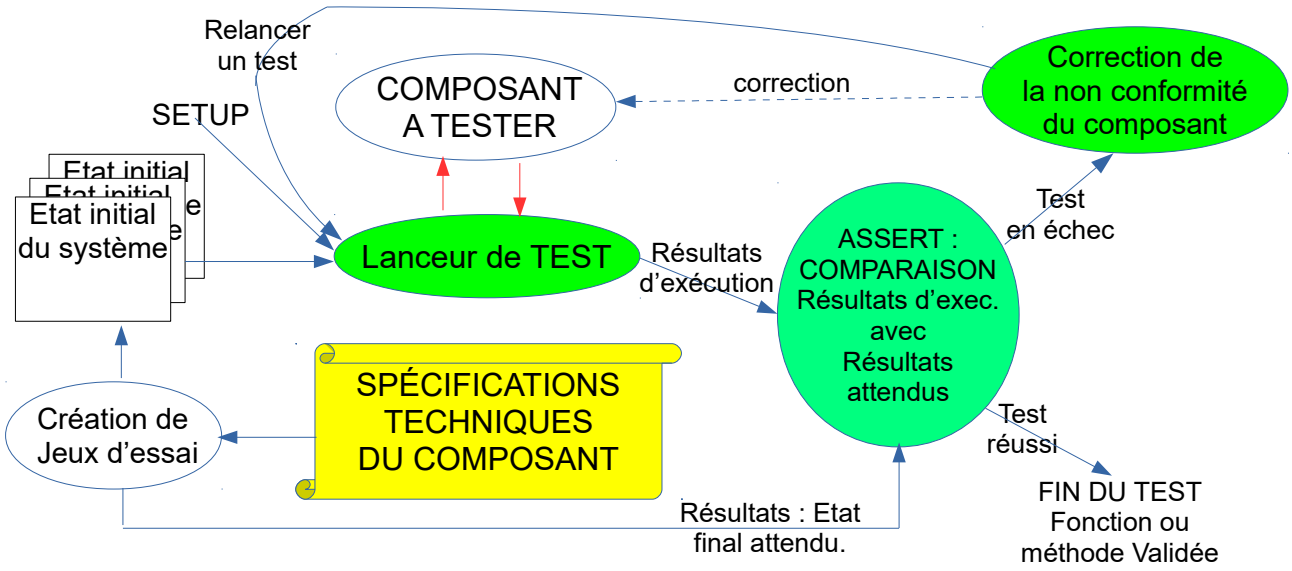
Si l'ÉCART existant entre les résultats attendus et les résultats d'exécution se situe en deçà de ce qui est acceptable, le test est RÉUSSI et la fonction ou la méthode est VALIDÉE. Sinon, elle est INVALIDÉE et son développement doit être repris pour corriger les non conformités constatées.

III.1.2.LISTE DES ACTIVITÉS:

- ***FIXURE:*** Cette activité définit un ENVIRONNEMENT DE DÉPART, COMPLET et REPRODUCTIBLE, correspondant à un environnement d'exécution prévu pour ce composant. FIXURE (terme anglais) peut se traduire par «attache»;
- ***SETUP:*** Cette activité définit l'ÉTAT INITIAL du composant à tester. En particulier, c'est cette activité qui va instancier la ou les classes utilisées par le test;
- Définition d'un ***CRITÈRE D'ARRÊT*** déclenchant la fin du test;
- ***EXERCICE:*** C'est l'activité de test proprement dite: elle consiste à statuer sur le succès ou l'échec du test, c'est à dire vérifier que l'état du système et en particulier des données de sortie du test correspondent bien à ce que les SPÉCIFICATIONS TECHNIQUES du composant testé permettaient de prévoir compte tenu des données d'entrée. Les fonctions concernées sont appelées FONCTIONS D'ASSERT;
- ***TEAR DOWN:*** cette activité consiste à restaurer l'environnement antérieur au test afin que d'autres tests effectués ultérieurement ne soient pas perturbés par des exécutions antérieures. TEAR DOWN se traduit par abattre, démolir.

Le schéma suivant représente l'enchaînement de ces différentes activités, dans le cadre d'une phase de CONCEPTION DÉTAILLEE.

Nous voyons sur ce schéma que l'activité de test est en général ITÉRATIVE: les non conformités mises à jour par les tests donnent lieu à une correction, puis à un nouveau test, jusqu'à ce que le composant soit entièrement conforme. Ce type de démarche est parfois appelé CONCEPTION GUIDÉE PAR LES TESTS.



COMMENTAIRES:

- Le Lanceur de TEST (Driver de test) est le logiciel informatique qui active la fonction ou la méthode du composant que l'on veut tester. Il lui transmet des paramètres de test qu'il extrait de JEUX DE TEST et récupère les paramètres retournés;
- un JEU D'ESSAI est une donnée (généralement un TABLEAU) qui collationne à la fois des DONNÉES D'ENTRÉE applicables à l'interface de la méthode ou de la fonction testée et les DONNÉES ATTENDUES en sortie du composant en fonction des spécifications techniques du composant et des valeurs des données d'entrée;
- L'action baptisée SETUP consiste à installer l'environnement d'essai convenant aux tests à effectuer;
- L'activation du composant par le test ne se réduit pas toujours à un appel procédural. Les flèches rouges du schéma peuvent représenter également un échange réseau, un envoi de signal, le déblocage d'un sémaphore, et toute interaction entre composants logiciels provoquant l'exécution de l'un d'eux;
- Les traitements de comparaison entre résultats attendues et résultats obtenus sont appelés FONCTIONS D'ASSERT (en anglais, to assert = affirmer)

III.1.3.OBJECTIFS D'UN TEST UNITAIRE:

Un test unitaire a pour objectif de tester L'APTITUDE du composant testé à satisfaire les EXIGENCES prévues dans les SPÉCIFICATIONS TECHNIQUES de ce composant.

Ces spécifications techniques peuvent concerner:

- Les EXIGENCES FONCTIONNELLES (capacité d'offrir aux utilisateurs les FONCTIONS prévues par les spécifications).
- Les EXIGENCES OPÉRATIONNELLES (Utilisabilité en environnement opérationnel, disponibilité, etc.);
- La FIABILITÉ (probabilité de défaillance sur une certaine période);
- Les PERFORMANCES exigées de ce composant (par exemple: durée d'exécution, précision des résultats), par rapport aux SPÉCIFICATIONS DE PERFORMANCES particulières du composant;
- La RÉSISTANCE AUX ERREURS (comportement face à des données d'entrées erronées ou illogiques (induisant par exemple une division par zéro).
- La QUALITÉ DE RÉALISATION (Conformité du code à une norme de codage, qualité de la documentation, etc.).

Très souvent, la littérature consacrée aux tests unitaires ne s'intéresse qu'aux TESTS FONCTIONNELS. Or, ceux-ci sont de loin les plus faciles à réaliser. Les autres catégories de test, qui concernent souvent des caractéristiques plus sensibles que le simple aspect fonctionnel (fiabilité, résistance aux erreurs, etc.), exigent souvent beaucoup plus d'inventivité.

III.1.4.SIMULATION DE L'ENVIRONNEMENT D'EXÉCUTION:

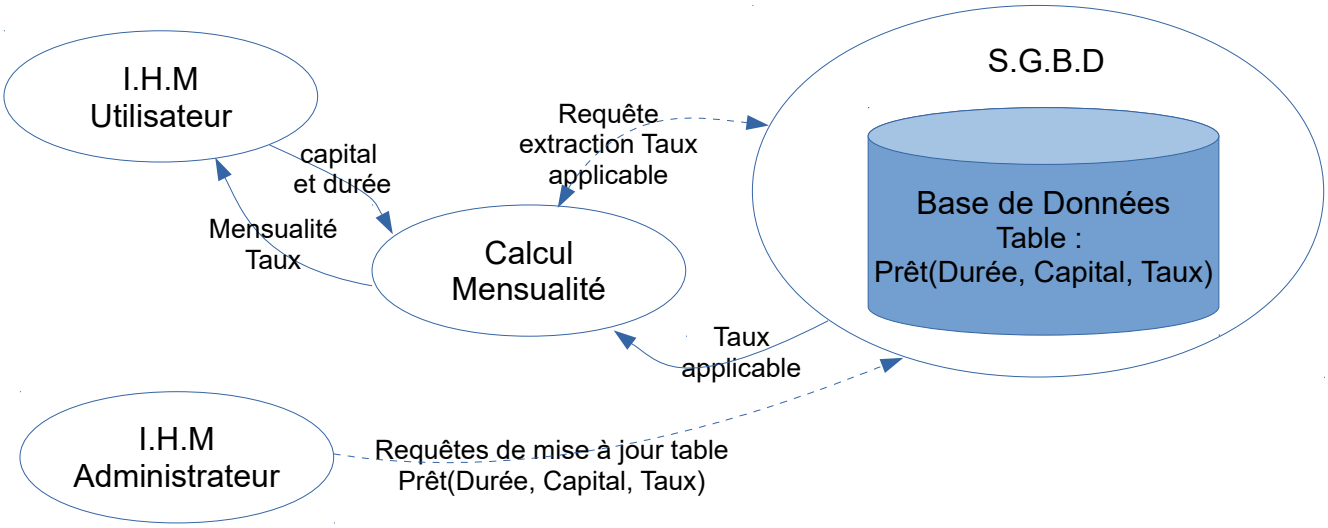
POSITION DU PROBLÈME:

La plupart du temps, un COMPOSANT est immergé dans un environnement opérationnel au sein duquel il interagit non seulement avec des composants **qui l'utilisent**, mais aussi avec des composants que **lui-même utilise** ou même avec des composants avec lesquels il **partage des ressources**.

EXEMPLE: le schéma suivant modélise un logiciel de calcul de prêt bancaire. Il fait apparaître 4 composants de premier niveau:

- l'IHM des utilisateurs: celui-ci permet de saisir le capital et la durée du prêt et d'afficher le montant de la mensualité correspondante et le taux d'intérêt applicable;
- l'IHM des administrateurs: celui-ci permet de mettre à jour une table de la base de donnée. Cette table, qui contient trois colonnes (Durée du prêt, Capital prêté, Taux d'intérêt applicable), permet de trouver, pour une durée et un montant donné, le taux d'intérêt consenti par la banque;
- Le composant CalculMensualité qui calcule les mensualités pour un capital de départ, une durée de prêt et un taux d'intérêts donnés.
- Le Système de Gestion de la Base de Données (S.G.B.D), muni d'une base de donnée contenant la table Prêt (Durée, Capital, Taux);

Le schéma suivant donne une représentation logique de ce logiciel:



Supposons que nous soyons chargés de tester unitairement le composant CalculMensualité. Ce composant peut se présenter sous la forme de la CLASSE ci-après:

Classe : CalculMensualité
Attribut privé Capital ; Attribut privé Durée; Méthode publique CalculerMensualité (val Capital, val Durée, ref Mensualité, ref Taux) ; Méthode privée ExtractionTauxApplicable (val Capital, val Durée) ;

CRÉATION DE JEUX D'ESSAI:

Un des tests unitaires les plus importants pour ce composant consiste évidemment à tester son aptitude à fournir au composant IHM Utilisateur le montant de la mensualité et le taux d'intérêt correspondant au montant du prêt et à la durée du prêt saisis par l'utilisateur sur l'IHM.

Il faudra donc en premier lieu que le driver de test active la méthode CalculerMensualité avec des valeurs représentatives pour les paramètres d'appel Montant et Durée de cette méthode. Le jeu d'essai pourra se présenter comme suit:

JEUX D'ESSAI			
VALEURS EN ENTRÉE		**RESULTATS ATTENDUES**	
CAPITAL (Euros)	**DURÉE MAXI (Mois)**	**TAUX (%)**	**MENSUALITÉ (Euros)**
1000	12	2,0	84
1000	24	4,0	43
1000	36	6,0	30
2000	12	1,5	168
2000	24	3,0	86
2000	36	4,5	59
Etc…			
Etc...			
10000	36	3,0	291
-3	3000	{ Valeurs aberrantes (pour tester la résistance aux erreurs).	
0	-700		
70000000000000	17,5		
Etc…			

REMARQUE:

La création de jeux d'essai assez complets pour tester complètement un composant complexe peut exiger une charge de travail très lourde par rapport au développement du composant lui-même. Il est donc important de développer des stratégies pour limiter le volume des jeux d'essai

sans dégrader leur efficacité:

- Lorsqu'il s'agit de tester des composants renfermant une logique interne complexe, une de ces stratégies consiste à choisir les valeurs d'entrée de façon à activer le plus possible tous les cas prévus par la logique interne du composant, mais en évitant de tester plusieurs fois le même cas.
- Lorsqu'un composant effectue des calculs numériques, il faut choisir les valeurs d'entrés en fonction des seuils de discontinuité des données (ici, passage d'un taux à un autre, par exemple). Dans l'exemple ci-dessus, on pourra choisir comme valeurs d'entrés l'ensemble des valeurs possibles du couple (Montant, Durée) avec Montant $\in$ { 1000, 2000, 3000, …, 10000 } et Durée $\in$ {12, 24, 36}.

Il est également très important d'inclure dans les entrées des VALEURS ABERRANTES: par exemple, un capital ou une durée négatifs ou nuls, une durée de 12000 mois, un capital de 30000000000 d'euros, des valeurs qui peuvent provoquer des divisions par zéro, etc. Ces valeurs mettent en évidences d'éventuelles FAILLES dans les traitements qui peuvent amener des résultats aberrants, la montée d'exceptions, des «plantages» de logiciels, ou même permettre des piratages par injection de code.

SIMULATION DE L'ENVIRONNEMENT OPÉRATIONNEL:

POURQUOI SIMULER L'ENVIRONNEMENT OPÉRATIONNEL?

Dans l'exemple que nous avons pris, il est évident que le composant CalculMensualité ne peut exécuter correctement sa méthode CalculerMensualité que s'il parvient à obtenir le Taux d'intérêt applicable pour un emprunt de la durée et du montant envisagé. En fonctionnement opérationnel, ce taux est le résultat d'une requête adressée à la base de donnée. Or, celle-ci est peut être développée par une autre équipe et pas encore disponible au moment du test unitaire.

Cette situation se présente dans pratiquement tous les cas où le composant testé n'est pas une simple bibliothèque de fonctions n'utilisant aucune ressource extérieure.

LES OUTILS DE SIMULATION DE L'ENVIRONNEMENT (STUBS, MOCKS, MUETS ou BOUCHONS):

Il faudra donc pallier l'indisponibilité des composants en relation avec le composant à tester en développant de «faux» composants capables de SIMULER au moins en partie le comportement des composants non disponibles. De tels composants sont souvent appelés STUBS (bouts) ou encore MOCKS (faux, contrefaits) en anglais. En Français, on emploie aussi les termes MUETS ou BOUCHONS pour les désigner. Ici, il faudra donc développer un composant simulant le comportement du SGBD lors de l'exécution des tests unitaires.

Pour que le composant à tester (ici, le composant CalculMensualité) puisse fonctionner normalement dans le contexte du test, il est nécessaire:

- Que le composant «simulateur» présente la même interface que le composant qu'il simule (c'est en particulier indispensable quand il doit être «LINKÉ» à la configuration de test, afin que l'édition des liens soit complète).
- Que les valeurs renvoyées par le composant simulé aient un sens dans le contexte d'exécution du composant à tester .

Il est possible de classer ces composants simulés en deux types:

1. **Les composants «passifs»**: ceux qui simulent un composant dont l'exécution n'influe pas sur l'exécution du composant à tester. Ce type de composant correspond aux appellations «STUB», «BOUCHON» ou encore «MUET»;
2. **Les composants «actifs»:** ceux qui, au contraire, simulent un composant dont l'exécution influe sur celle du composant à tester. C'est à ce type de composants que s'applique l'appellation «MOCK».

REMARQUE N°1: le composant à simuler peut influer sur le composant a tester de diverses manières:

- Par les paramètres qu'il retourne au composant à tester;
- Par les modifications qu'il apporte à l'environnement d'exécution: réservation ou libération d'une ressource commune, armement d'un événement ou d'une temporisation utilisée par le composant à tester;

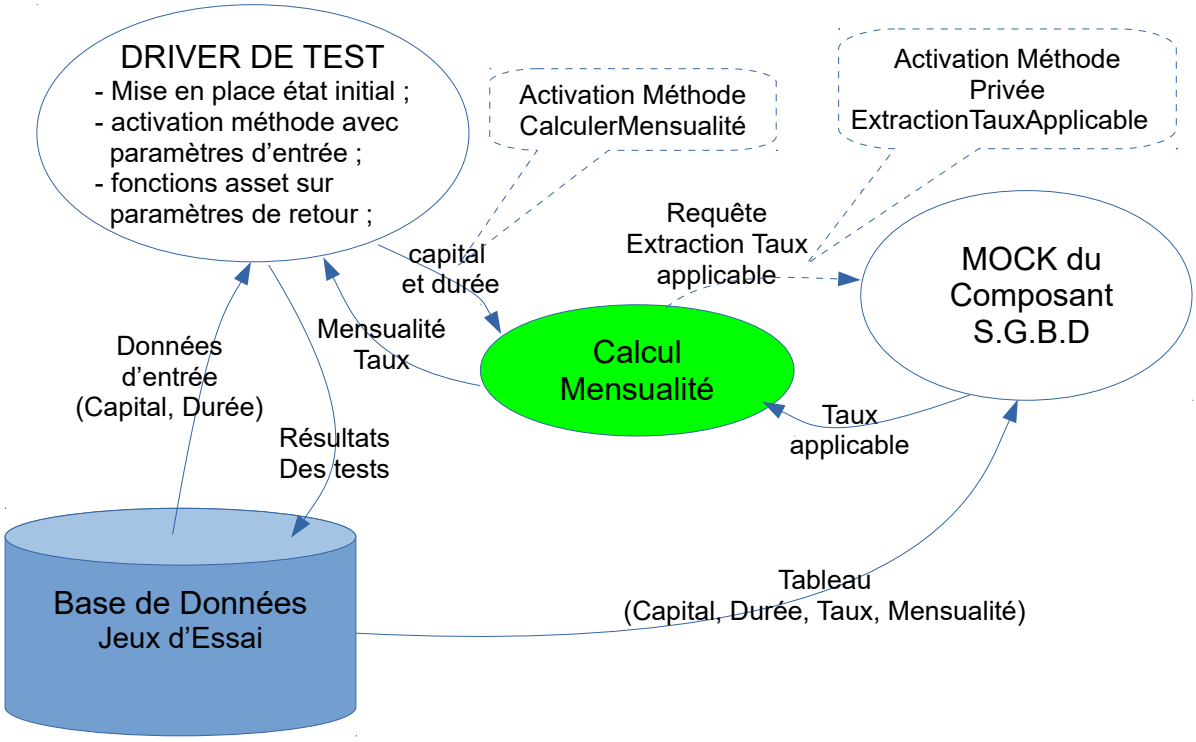
- Sa durée d'exécution;
- etc.

De ce fait, les MOCKS encapsulent en général une algorithmique complexe destinée à simuler le comportement du composant qu'ils remplacent et les interactions de tous types qu'ils retournent à l'objet à tester (arguments de retour, exceptions logicielles, signaux, etc.).

EXEMPLE: le MOCK destiné à simuler le comportement du composant SGBD pourra encapsuler le tableau représentant le jeux d'essai et, pour une valeur donnée du capital et de la durée de prêt, retournera au composant à tester le montant des mensualité et le taux d'intérêt appliqué en utilisant ce tableau. Il pourra également simuler la durée de la transaction avec la base de donnée en ralentissant son exécution à l'aide d'une temporisation programmée représentative de cette transaction.

*III.1.5.*CONFIGURATION DE TEST DU COMPOSANT CALCUL MENSUALITÉ:

La configuration de test de la méthode CalculerMensualité du composant CalculMensualité peut être représentée comme suit:



RÉCUPÉRATION DES RÉSULTATS DE TEST:

Dans l'exemple présenté ci-dessus:

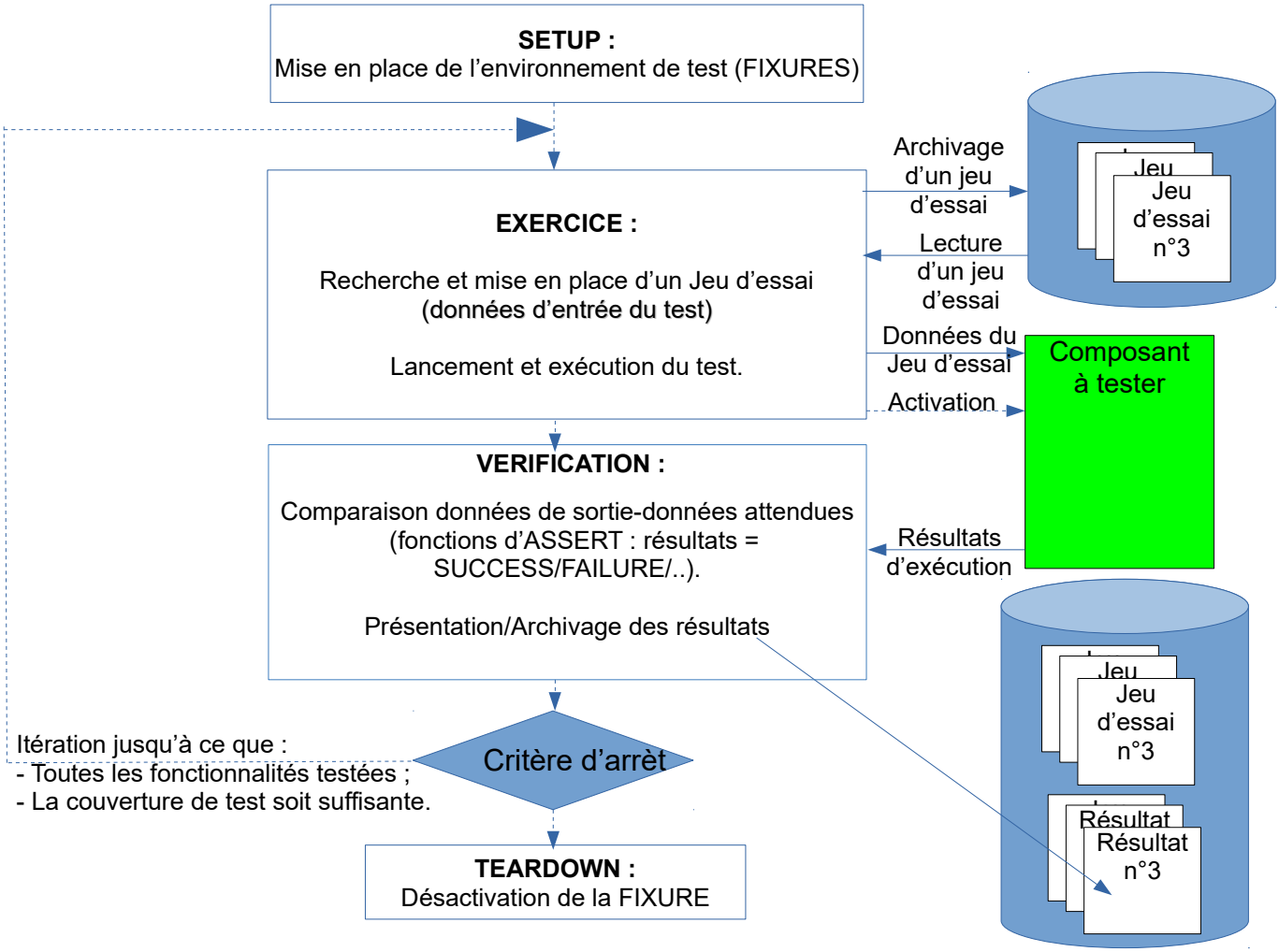
- Le DRIVER DE TEST va lire le jeu d'essai ligne par ligne. Pour chaque ligne, il va extraire et mettre en place l'état initial prévu par le jeu d'essai (c'est la fonction SETUP);
- Puis, il va activer la méthode CalculerMensualité avec les arguments Capital et Durée contenus dans chaque ligne (c'est la fonction EXERCICE);
- Après exécution de la méthode, il va récupérer les arguments de retour Taux et Mensualité et comparer ceux-ci aux valeurs prévues par le jeu d'essai (c'est la fonction VÉRIFICATION qui va activer les fonctions d'«assert» pour déterminer si le test est concluant ou non pour la ligne traité);
- Enfin, le driver va récupérer ces résultats, en faire une synthèse et archiver le tout (par exemple dans un fichier «log»);
- A la fin des test, les résultats pourront être fournis de diverses manières.

REMARQUE: Si le composant testé a été INSTRUMENTÉ pour évaluer la COUVERTURE du test, cette couverture pourra être également sauvegardée avec les résultats: elle permettra de compléter le jeux d'essai afin d'améliorer cette couverture. L'évaluation de la couverture des test est abordée au chapitre suivant.

III.1.6.SCHÉMA DE PRINCIPE:

Le schéma ci-dessus représente l'algorithme général d'un test unitaire automatisé, avec archivage des résultats des tests.

Les lignes pleines représentent les flux de données, les lignes pointillées représentent les flux de commandes. Les rectangles représentent les blocs de traitements.



Un logiciel réalisant cet algorithme est souvent appelé DRIVER (driver de test) ou LANCEUR DE TEST en Français.

Comme nous l'évoquions plus haut, un test unitaire ne peut pas toujours être entièrement automatisé, mais la procédure générale se conforme toujours à peu près à l'algorithme représenté.

III.1.7.RECOMMANDATIONS DIVERSES:

- La procédure de test d'un composant doit être relativement peu complexe par rapport au composant à tester. En effet, le testeur doit avoir confiance en la validité des procédures de test qu'il utilise. Or, plus une procédure de test est complexe, plus il est difficile de valider son fonctionnement sans développer une «procédure de test du test». De ce fait, il est recommandé de ne tester qu'une seule exigence sur l'objet testé (ou même une seule caractéristique de cette exigence);
- Un test unitaire d'aptitude est de type «BOÎTE NOIRE»: ceci veut dire que ce test ne se préoccupera que des commandes et des données ENTRANT ou SORTANT du composant. Ainsi, si le composant est une CLASSE, on ne testera que les méthodes PUBLIQUES;
- Le développement d'un test unitaire est principalement contraint par les exigences à tester. De ce fait, dés la rédaction des exigences portant sur le module à tester, il est primordial de veiller à ce que celles-ci soient rédigées de façon à être TESTABLES. Une exigence est dite «testable» s'il est possible de vérifier d'une manière simple et sans ambiguïté que cette exigence est satisfaite ou au contraire qu'elle n'est pas remplie.

  Exemple: *l'exigence «l'incertitude sur la valeur des données mesurés doit être inférieure ou égale à 0,1%» est testable alors que l'exigence «la valeur des données mesurés doit être la plus précise possible» ne l'est pas.*

IV.ÉVALUATION DE LA COUVERTURE D'UN TEST:

IV.1.COUVERTURE D'UN TEST AU SENS GÉNÉRAL:

La «COUVERTURE» d'un test est une valeur numérique qui reflète dans quelle mesure un test a permis de valider le fonctionnement d'un composant. En général, cette valeur s'exprime par un POURCENTAGE.

EXEMPLE:

Lors de la livraison d'un logiciel, la documentation technique de l'application peut être considérée comme un composant de la fourniture. Ce composant doit donc être vérifié avant livraison. A un moment donné de la procédure, si le test a été effectué sur 70% du texte: on pourra dire que la couverture du test est de 70%. Elle sera de 100% lorsque l'intégralité du texte aura été vérifiée.

IV.2.COUVERTURE DE CODE:

IV.2.1.DÉFINITION:

La COUVERTURE DE CODE s'applique aux CODES OBJETS d'une application, qui sont des textes. Le fait qu'un COMPOSANT ait subit une série de tests CONCLUANTS, concernant toutes ses APTITUDES, n'implique pas forcément que cette série de tests ait activé la totalité du code objet de ce composant. En effet:

- Certaines fonctionnalité «consomment» un grand nombre de données d'entrée, qui elles-mêmes peuvent prendre un grand nombre d'états: dans ce cas, il est très difficile de garantir que toutes les configurations d'entrée, toutes les combinaisons de valeurs de paramètres ont été prises en compte dans les jeux d'essai;
- En particulier, toutes les configurations «illogiques», résultant par exemple de paramétrages aberrants, doivent être testées: dans un logiciel de qualité, ces configurations sont normalement «bloquées» par le logiciel, mais il importe de s'en assurer;
- Un code objet peut contenir des parties de codes qui sont inatteignables quelle que soit la configuration des entrées: ces codes correspondent en général à des erreurs de programmation où à des fonctions dont l'implémentation a été abandonnée en cours de développement (on les appelle souvent «codes morts»). Laisser un code mort dans une application, outre qu'il bloque une partie de la mémoire, peut être dangereux: en effet, rien ne ressemble plus à un code mort qu'une «bombe logicielle», destinée à s'activer dans une configuration rarissime (exemple: le premier janvier 2024 à minuit) ou qu'un logiciel espion;
- Remarquons qu'il est possible que certaines zones du code objet ne puissent pas être atteintes par des appels procéduraux. C'est par exemple le cas des contrôleurs d'événements ou d'interruptions qui ne sont activés que par ces événements ou interruptions. Pour obtenir une couverture de code complète, il faudra dans ces cas imaginer une configuration de test qui provoque ou simule ces événements ou interruptions.

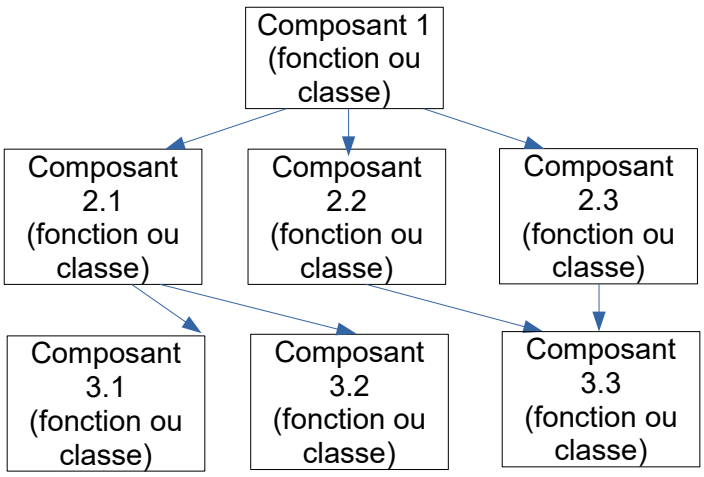
IV.2.2.DIFFÉRENTS NIVEAUX DE TESTS DE COUVERTURE DE CODE:

Le tableau ci-après énumère et décrit, dans l'ordre de complexité croissant, les 4 principaux niveaux de couverture de code:

DÉSIGNATION	DESCRIPTION
Couverture des fonctions (Function Coverage)	On mesure le pourcentage de fonctions ou de méthodes du logiciel qui ont été activées et vérifiées par les tests.
Couverture des instructions (Statement Coverage)	On mesure le pourcentage d'instructions du code objet qui ont été exécutée et vérifiée par les tests.
Couverture des points de tests (Condition Coverage)	On mesure le pourcentage des ALTERNATIVES LOGIQUES contenues dans le code objet qui ont été activées et vérifiées par les tests. Ces alternatives logiques peuvent par exemple résulter du test de la valeur d'une variable (est-elle plus grande, égale ou plus petite que zéro?).
Couverture des chemins d'exécution (Path Coverage)	Elle mesure le pourcentage des CHEMINS D'EXÉCUTION possibles qui ont été parcourus lors des tests. ***REMARQUE:*** l'ensemble des chemins d'exécution possible résulte de la combinaison de toutes les conditions possibles résultant de l'activation des points de test: certains chemins peuvent donc être inatteignables.

COUVERTURE DES FONCTIONS:

Évaluer la couverture des fonctions revient à mesurer le pourcentage de fonctions ou de méthodes du logiciel qui ont été activées et vérifiées par les tests. Un logiciel conçu de façon MODULAIRE peut être représenté par le diagramme ci-dessous appelé diagramme des appels:



Les flêches représentent les flux de contrôle entre les composants (appels de fonctions ou de méthodes).

La couverture des fonctions consiste à vérifier le pourcentage des fonctions ou méthodes que les tests ont activées.

NOTA: *Il s'agit, bien sûr, de tests de couverture très sommaires, puisqu'on considère les composants comme des «boîtes noires»: la couverture des fonction peut être de 100% alors que de nombreux éléments de chaque composant n'ont pas été activés.*

NOTION DE SÉQUENCES D'INSTRUCTIONS:

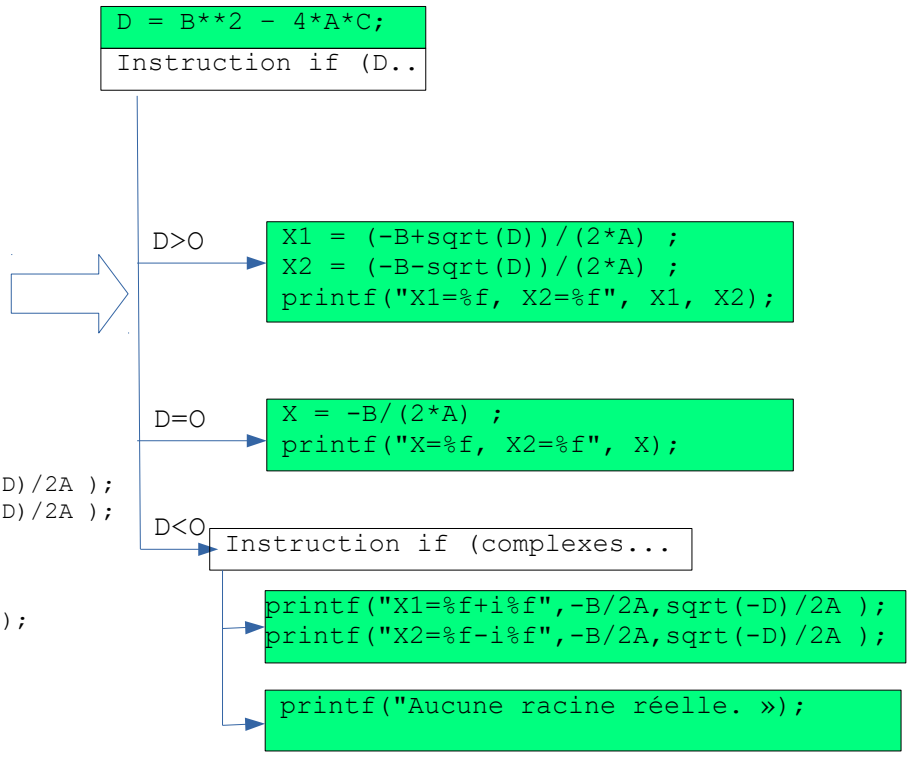
Un code objet se présente comme un texte dans lequel les instructions du langage sont écrites les unes après les autres, formant une «séquence» d'instructions. Si nous considérons l'ordre dans lequel ces instructions doivent être exécutées, nous pouvons distinguer deux types d'instructions:

- **Type 1:** Celles qui n'interrompent pas la séquence des instructions écrites (dans ce cas, l'instruction qui sera exécutée après l'instruction en question sera la suivante dans la suite des instructions écrites);
- **Type 2:** Celles qui, après leur exécution, interrompent la séquence écrite pour transférer l'exécution à un autre endroit du code (rupture de séquence).

Les instructions de type 2 sont les instructions qui testent des conditions (instructions «if… then ...else», «if… then ...else if … else...», «alternatives multiples: case, switch..» . Du point de vue de l'exécution, ces instructions séparent le code source en séquences d'instructions de type 1:

Exemple:

```
D = B**2 - 4*A*C;
if ( D > 0 )
{
   X1 = (-B+sqrt(D))/(2*A) ;
   X2 = (-B-sqrt(D))/(2*A) ;
   printf("X1=%f, X2=%f", X1, X2);
}
else if ( D = 0 )
{
   X = -B/(2*A) ;
   printf("X=%f, X2=%f", X);
}
else
{
   if ( complexes == true )
   {
       printf("X1=%f+i%f",-B/2A,sqrt(-D)/2A );
       printf("X2=%f-i%f",-B/2A,sqrt(-D)/2A );
   }
   else
   {
       printf("Aucune racine réelle. »);
   }
}
```



Nous pouvons constater ici que les deux instructions de type 2 if … else partagent le code en cinq séquences d'instructions de type 1 (en vert).

COUVERTURE DES INSTRUCTIONS:

Elle consiste à évaluer le pourcentage des instructions du code objet qui ont été activées par les tests.

Un test de couverture de ce type peut être réalisé en intercalant au début de toutes les séquences du code (définies comme dans le paragraphe précédent) l'appel à une fonction (ou une méthode) qui enregistre la «trace» du passage du flux de contrôle à cet endroit du code. Cette opération est souvent nommée INSTRUMENTATION DU CODE

Par principe, si l'instruction de début d'une séquence est exécutées, toutes les autres instructions de la séquence le sont. Par ce moyen, on pourra donc vérifier qu'une instruction quelconque de la séquence a été exécutée et combien de fois elle l'a été.

RÉALISATION SANS OUTIL:

- Dans un premier temps, le code sera séparé en séquences, comme il est indiqué au paragraphe précédent. Toutes les séquences recevront un identificateur unique;
- Puis, les appels à la fonction d'enregistrement seront intercalés en début de chaque séquence. L'identificateur de cette séquence sera passé en paramètre;
- Les tests seront alors déroulés avec le code instrumenté;
- A la fin des tests, les enregistrements seront exploités de façon à visualiser, pour chaque séquence, le nombre de fois qu'elle a été activée et le pourcentage total des instructions activées;
- Si l'on constate qu'une séquence n'est jamais activée, il suffit d'ajouter un jeu de test adéquat (ou de simuler l'événement ou l'interruption qui active cette séquence) et de refaire les test. Si cette opération ne peut être réalisée, il s'agit probablement d'un code mort.

REMARQUE: *L'instrumentation du code peut être prévue dès la phase de codage initial. Pour éviter que cette instrumentation ne perturbe l'exécution en phase opérationnelle, il suffit d'utiliser des directives de compilation conditionnelles, quand le langage le permet.*

RÉALISATION AVEC UN OUTIL LOGICIEL:

L'instrumentation d'un code «à la main» est, bien sûr, une opération fastidieuse et chronophage pour les développeurs. Heureusement, de nombreux outils logiciels réalisent automatiquement ces opérations.

COUVERTURE DES «POINTS DE TEST»:

La notion de «point de test» désigne les instructions pouvant provoquer une rupture de séquence sous condition. A un point de test peuvent être associées deux séquences de code pouvant être exécutées alternativement.

Le point de test le plus simple est défini par les instructions de type «if … then … else ...». Les structures multiples de type «if … then … else if … then… else if …else … » ou les structures de type «switch … case …. case …» peuvent être vues comme des successions d'alternatives. Par exemple, un «switch» muni de 5 «case» définit 5 «points de test».

La couverture des points de test correspond à une exigence supérieure en matière de couverture de code: on essaie d'évaluer si toutes les alternatives ont été activées par les test. En effet, il est des cas où la couverture de toutes les instructions n'implique pas que toutes les alternatives ont été activées:

Exemple:

```
Soit le fragment de code suivant:
printf(" ");
if (val < 0)
{
   val = -val;
}
R = sqrt( val );
printf("La racine carrée du module de val est %f", R );
```

Ce fragment contient 3 séquences. Si val est plus petit que zéro, toutes les instructions de toutes les séquences sont parcourues (couverture des instructions = 100%). En revanche, une seule des alternatives du «if» a été activée (couverture des points de test = 50%): on n'a donc pas testé le logiciel pour l'alternative val ≥ 0.

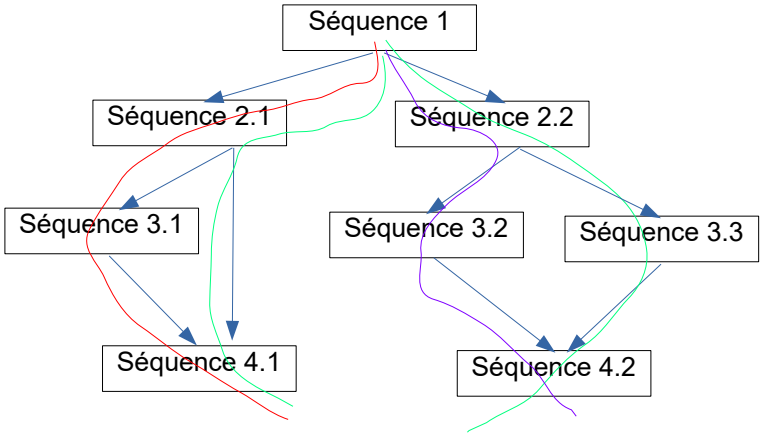
De même, la couverture des points de test n'implique pas forcément la couverture des instructions, car un fragment de code peut très bien ne correspondre à aucune alternative (code «mort»).

RÉALISATION:

La réalisation exige, outre l'instrumentation du code comme précédemment, d'identifier pour tous les points de test toutes le alternatives possibles. Un tel travail ne peut être raisonnablement réalisé «à la main». Il faut donc utiliser un outil logiciel.

COUVERTURE DES CHEMINS:

Un CHEMIN D'EXÉCUTION peut être représenté par la concaténation de toutes les séquences de code exécutées si à chaque point de test, on choisit une des alternatives:



Exemple :
Le code représenté ci-contre comprend 3 points de test qui définissent 4 chemins.

La couverture complète des chemins d'un code objet est toujours très difficile à réaliser, surtout lorsque la logique de ce code est complexe. Elle peut même être impossible car:

- La logique de l'algorithme peut interdire certains parcours (pour raison de sécurité, par exemple);
- Les opérations itératives (boucles) peuvent aboutir à un nombre indéterminable de chemins possibles. De ce fait, on ne tient pas compte dans les chemins des instructions itératives: on ne teste que le «chemin de base» correspondant à un seul parcours de chaque itération.

IV.2.3.LA MÉTHODE MC/DC:

MC/DC est l'abréviation de l'expression Modified Condition/Decision Coverage. Cette méthode est recommandée par la norme de certification DO-178B5 lorsqu'il s'agit de tester des logiciels soumis à de très hauts critères de sécurité. La validation MC/DC implique que toutes les exigences ci-dessous soient respectées au moins une fois:

- Pour chaque décision (point de test), toutes les issues possibles sont essayées;
- Chaque condition, dans une décision, prend toutes les issues possibles ;
- Chaque point d'entrée et de sortie est passé ;
- Chaque condition dans une décision affecte indépendamment l'issue de la décision.

IV.2.4.OUTILS DE COUVERTURE DE CODE:

OUTIL	ENVIRONNEMENT	DETAIL
Testwell CTC++	C, C++, Java et C#	• Compatible MC/DC (norme de certification DO-178B5); • Disponible sous Windows, Linux, Solaris et HP-UX; • Pour systèmes embarqués.
LDRA	C, C++, Java et Ada	• Analyse de conformité du code et de couverture du code; • Compatible MC/DC (norme de certification DO-178B5);
Xdebug	PHP	Extension de PHP incluant des fonctions de débogage et de couverture de code.

V.PLACE DES TESTS UNITAIRES DANS LE PROCESSUS DE DÉVELOPPEMENT:

V.1.QUAND FAUT-IL ÉLABORER LES TESTS?

V.1.1.CONTRAINTES TECHNIQUES:

Les tests unitaires d'un COMPOSANT d'un logiciel ne peuvent être développés directement à partir des SPÉCIFICATIONS TECHNIQUES DE BESOIN (S.T.B) de ce logiciel. En effet, à ce niveau d'avancement d'un projet, l'ARCHITECTURE LOGICIELLE n'étant pas encore définie, les COMPOSANTS de cette architecture ne le sont évidemment pas.

La S.T.B permet donc tout au plus d'élaborer les TESTS DE RÉCEPTION (encore appelés TESTS DE VALIDATION) du logiciel complet.

Pour élaborer les tests unitaires d'un composant, il faut disposer des SPÉCIFICATIONS TECHNIQUES PARTICULIÈRES (S.T.P) de ce composant, qui ne peuvent être élaborées que pendant la phase de CONCEPTION PRÉLIMINAIRE (après élaboration de l'architecture logicielle globale).

Les tests unitaires d'un composant ne peuvent donc être élaborés qu'APRÈS la phase de CONCEPTION PRÉLIMINAIRE.

REMARQUE: *Ceci contredit apparemment beaucoup d'écrits sur le sujet qui conseillent d'élaborer les tests immédiatement après la phase de «spécifications». Je pense que la contradiction n'est qu'apparente et que les auteurs se réfèrent implicitement aux spécifications de* ***l'entité à tester****.*

V.1.2.ÉCRIRE LES TESTS AVANT L'OBJET A TESTER:

Les démarche Test-Driven Development (TDD) (développement piloté par les tests) et XP (Extrême Programming – méthode agile) préconisent d'écrire les tests unitaires AVANT le code source du composant. Cette option présente plusieurs avantages:

- Elle permet de définir précisément l'interface du module à développer. Les tests peuvent être exécutés durant tout le développement du composant. Ils permettent ainsi de vérifier que les versions successives du code sont et restent conformes à la STP (Spécifications Techniques Particulières) du composant et qu'il n'y a pas de régression entre deux itérations du développement;
- Les tests unitaires sont de précieux complément à la S.T.P: leur lecture peut permettre de préciser comment s'utilise une fonction ou une méthode: la documentation peut ne pas être à jour alors que les tests correspondent forcément à l'état de l'application.

V.2.LA DÉMARCHE TEST-DRIVEN DEVELOPMENT (TDD)

TDD préconise d'écrire les tests unitaires AVANT le code source du composant. Elle propose également un cycle de développement itératif combiné des tests et du composant comportant cinq étapes:

1. Écrire une première version du logiciel de test (assez simple pour ne pas nécessiter d'être elle-même testée);
2. Vérifier que le test échoue: ceci est normal puisque le composant à tester n'existe pas encore. Cela permet de vérifier que le résultat est non valide dans le cas où le composant à tester ne répond pas correctement;
3. Écrire une première version du composant à tester avec un code minimal mais suffisant pour passer le test (c'est à dire le code qui supporte les traitements testés par le logiciel de test) ;
4. Vérifier que le test réussit pour cette version de ce composant;
5. Si le test ne réussit pas, corriger le code du composant à tester, puis revenir en 3 jusqu'à ce que le test passe;
6. Si le test réussit, améliorer le code du composant à tester (sans ajouter de fonctionnalité), puis revenir en 4;
7. L'itération s'arrête lorsque le composant aura atteint une maturité suffisante (concernant la fonction testée) pour le client et les développeurs.

Par sa structure itérative, TDD s'intègre très facilement à une démarche de développement AGILE. Cependant, cette démarche de développement conjoint du test et de l'objet à tester risque d'aboutir à des architectures résultant de retouches successives, donc peu optimisées.

VI.LES FRAMEWORKS DE TESTS UNITAIRES:

VI.1.HISTORIQUE ET GÉNÉRALITÉS:

Le langage Smalltalk (langage objet-1972) s'est enrichi en 1994 d'outils de tests unitaires (SUnit1).

En 1997 apparaît le framework JUnit de test unitaire pour le langage Java.

Celui-ci connaît un grand succès et entraîne la création de nombreux frameworks de tests unitaires. Ces tests ont le plus souvent un nom du type xUnit (x représentant le langage de programmation: JUnit pour java, PHPUnit pour PHP…).

En général, les environnements de test offrent uniquement:

- Un CADRE DE TRAVAIL (FRAMEWORK) qui permet de développer les tests d'une manière méthodique et rationnelle;
- Une bibliothèque de fonctions d'ASSERT;
- Une application qui permet, par une «introspection» du code des tests de repérer les différentes parties des tests (fonctions de FIXURE, SETUP, EXERCICE, TEARDOWN, développées par le tester), puis de les enchaîner afin de réaliser les tests et d'afficher les résultats.

Les corps des fonctions de fixure, setup, exercice et teardown doivent donc être développés spécifiquement en fonction des traitements à tester.

VI.2.LE FRAMEWORK JUNIT:

VI.2.1.PRÉSENTATION:

JUNIT est un outil open source développé par Erich Gamma et Kent Beck. Associé au langage JAVA, il permet la création de tests unitaires automatisables. Les objets testés par JUNIT sont des CLASSES java.

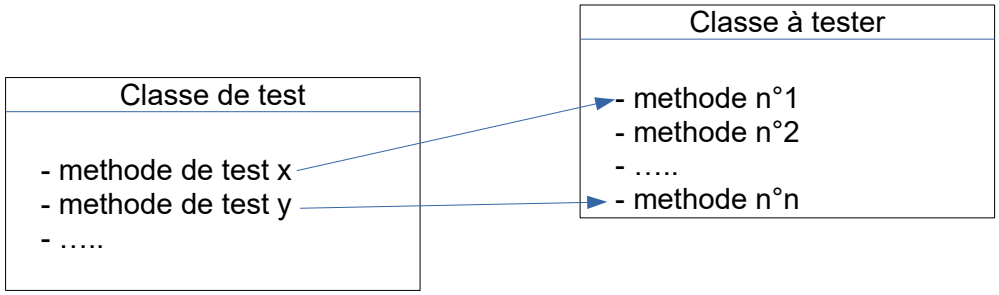
VI.2.2.STRUCTURE GÉNÉRALE:

JUNIT est composé:

- D'un CADRE DE TRAVAIL fixant les règles d'architecture et de fonctionnement des tests;
- D'une bibliothèque de fonctions d'assert;
- D'une application repérant par introspection des codes de test les différentes parties des tests (fonctions de FIXURE, SETUP, EXERCICE, TEARDOWN), puis enchaînant leur exécution et affichant les résultats.

Le logiciel JUNIT peut être utilisé en ligne de commande. Il est également intégré à des ateliers de développement JAVA comme ECLIPSE: dans ce cas, il est géré par une interface graphique.

Chaque TEST correspond à une classe et renferme un certain nombre de MÉTHODES DE TEST. Chacune de ces méthodes de test correspond à un CAS DE TEST et permet de tester une méthode de la classe à tester:



JUnit assure l'exécution de ces tests et la comparaison avec les résultats attendu (grâce aux fonctions assert) .

L'avantage de cette organisation est d'assurer une séparation complète entre le code de la classe à tester et celui de la classe de test. Les cas de tests peuvent être exécutés individuellement ou sous la forme de suites de tests.

REMARQUE: JUnit est particulièrement adapté à la méthode Xtreme Programming. En effet, cette méthode préconise le développement des tests AVANT l'objet à tester et l'utilisation des tests tout au long du développement (développement guidé par les tests).

VI.2.3.ORGANISATION DU CODE DE TEST:

JUnit définit deux sortes de CLASSES:

- Les classes de type TESTCASE qui contiennent un certain nombre de méthodes de tests (cas de test), chacune servant à tester le bon fonctionnement d'une méthode d'un composant (dans java, un composant est forcément d'une classe, donc, il s'agit de tester une méthode);
- Les classes de type TESTSUITE qui permettent d'exécuter des TESTCASE déjà définis et donc, d'enchaîner les tests automatiquement.

Chaque CAS DE TEST exécute les tâches suivantes :

- Création d'une instance de la CLASSE A TESTER et de toutes les dépendances nécessaires (y compris les MOCKS nécessaires à l'exécution;
- Appel de la méthode à tester avec les paramètres du jeu de test à utiliser;
- Comparaison du résultat attendu avec le résultat obtenu : en cas d'échec, une exception est levée

REMARQUE: *Dans un TestCase il n'y a pas de méthode «main»: chaque méthode de test est indépendante des autres.*

VI.2.4.ORGANISATION D'UNE CLASSE TESTCASE:

Une telle classe hérite de junit.framework.TestCase. Elle possède 5 types de méthodes, décrites dans le tableau ci-dessous:

Noms de méthodes	Appel	Fonctionnement
setUpClass	Appelée avant toute autre méthode.	Met en place l'environnement de test commun à tous les tests de la classe.
setUp	Appelée avant toute méthode «test»	Met en place l'environnement de test spécifique à un cas de test donné.
test<nom du cas de test> (c'est la méthode de test proprement dite) *Ex:* ***testAddition*** *teste le cas de test «Addition».*	Normalement appelé entre deux méthodes setUp et tearDown, si elles existent.	Exécute un cas de test. Active la méthodes à tester avec des paramètres d'entrée correspondant aux jeux d'essai, puis vérifie le bon comportement des méthodes testées en utilisant des méthodes internes ASSERT. Le résultat des différentes assertions (SUCCÈS, FAILURE, etc.) est mémorisé.
tearDown	Appelée après une méthode «test»	Annule l'environnement mis en place avant un test (par setUp).
tearDownClass	Appelée après toute autre méthode.	Annule l'environnement mis en place avant tous les tests de la classe (par setUpClass).

Une méthode de test proprement dite (celle qui correspond à l'activité EXERCICE du test) est caractérisée par:

- Son nom doit commencer par la chaîne "test" en minuscule, suivie du nom du cas de tests (on le prend en général identique nom de la méthode à tester);
- Elle doit être déclarée publique
- Elle ne doit renvoyer aucune valeur
- Elle ne doit pas transmettre de paramètre à l'appelant.

VI.2.5.LES FONCTIONS ASSERT:

Les cas de tests utilisent des affirmations (assertion en anglais) sous la forme de méthodes nommées assertXXX() proposées par le framework. Ces méthodes sont héritées de la classe junit.framework.Assert :

MÉTHODE	RÔLE
assertEquals()	Vérifier l'égalité de deux valeurs de type primitif ou objet (en utilisant la méthode equals()). Il existe de nombreuses surcharges de cette méthode pour chaque type primitif, pour un objet de type Object et pour un objet de type String
assertFalse()	Vérifier que la valeur fournie en paramètre est fausse
assertNull()	Vérifier que l'objet fourni en paramètre est null
assertNotNull()	Vérifier que l'objet fourni en paramètre n'est pas null
AssertSame() AssertTrue()	Vérifier que les deux objets fournis en paramètre font référence à la même entité assertSame("Les deux objets sont identiques", obj1, obj2); assertTrue("Les deux objets sont identiques ", obj1 == obj2);
assertNotSame()	Vérifier que les deux objets fournis en paramètre ne font pas référence à la même entité

En général, le premier argument des fonctions d'assert est une chaîne de caractères qui permet de repérer le cas de test dans les messages signalant les erreurs.

Il est, bien sûr, possible de n'utiliser que la méthode assertTrue(). Cependant, l'utilisation des autres méthodes permet d'alléger l'écriture des cas de test.

VI.2.6.STRUCTURE DU CODE D'UNE CLASSE TESTCLASS:

Le squelette de code objet ci-dessous correspond à celui d'une testClass:

```
import junit.framework.TestCase;
import org.junit.*;
public class NomDeMaClass extends TestCase
{
   @BeforeClass
   public static void setUpClass() throws Exception
   {
      // Code à exécuté avant tout autre méthode
   }
   @AfterClass
   public static void tearDownClass() throws Exception
   {
      // Code exécuté après l'exécution de toutes les autres méthodes
   }
   @Before
   public void setUp() throws Exception
   {
      // Code exécuté avant chaque test
   }
   @After
   public void tearDown() throws Exception
   {
      // Code exécuté après chaque test
   }
   @Test
   public void test1()
   {
     // code qui teste une méthode du composant à tester
     // Appelle cette méthodes avec les arguments prévus par le jeu d'essai
     // Active une assertion pour vérifier si une condition sur le comportement de
     // la méthode testée est vraie ou fausse.
   }
   @Test
   public void test2()
   {
     // code qui teste une autre méthode du composant à tester
     // Appelle cette méthodes avec les arguments prévus par le jeu d'essai
     // Active une assertion pour vérifier si une condition sur le comportement de
     // la méthode testée est vraie ou fausse.
   }
   // Etc……pour toutes les méthodes de test
}
```

VI.2.7.EXÉCUTION DES TESTS:

Rappelons les caractéristiques d'une méthode testant un cas de test:

- Son nom doit commencer par la chaîne "test" en minuscule, suivie du nom du cas de tests (on le prend en général identique nom de la méthode à tester);
- Elle doit être déclarée publique
- Elle ne doit renvoyer aucune valeur
- Elle ne doit pas transmettre de paramètre à l'appelant.

Les caractéristiques énoncées ci-dessus qui permettent au logiciel JUNIT de repérer ces méthodes (par «introspection» du code). Le logiciel JUNIT repère également dans chacune des classes TESTCASE les méthodes SetupClass, Setup, Teardown et TeardownClass.

Ceci lui permet d'enchaîner les tests sans qu'il soit nécessaire de définir une méthode «main» ou un constructeur (du moins jusqu'à la version 3.7 de JUnit, pour laquelle un constructeur passant une chaîne en paramètre est obligatoire).

Les méthodes de test déclarées dans une classe test case sont exécutées les unes après les autres. Dès qu'un test échoue, l'exécution de la méthode correspondante est interrompue, les résultats sont affichés et JUnit passe à la méthode suivante.

Les méthodes de test (de type test<nom du cas de test> contiennent au minimum les traitements suivants:

- La classe contenant la méthode à tester est instanciée (si cela n'a pas été fait par la classe SetUp ou SetupClass);
- La méthode à tester est appelée avec des paramètres de test;
- Les résultats obtenus sont comparés aux résultats attendus en utilisant une méthode assert.

Exemple:

```
public void testDivision throw exception
{
        Calcul Calc = new Calcul();             // Instancie la classe à tester (si ce n'est pas fait
                                                // avant)
        double R = Calc.Diviser ( 4, 5 );       // Exécute la méthode à tester avec les valeurs
                                                // d'entrée 4 et 5
        assertEquals( "Test Additionner-4+5", (double) 0.8 , R );  // Vérifie que le résultat est bien
                                                                   // égal à la valeur attendue.
                                                                   // (4/5 => 0.8=)
}
```

REMARQUE: il est tout à fait possible d'introduire une boucle de traitement afin de tester une collection de valeurs de paramètres d'entrée. Cependant, la première erreur détectée interrompt cette boucle.

VI.2.8.EXEMPLE DÉTAILLÉ:

Classes à tester:

```
/*
* Classe AdSous
*/
public class AdSous
{
	public double a;
	public double b;


	public double Additionner ( double x, double y)
	{
		this.a = x;
		this.b = y;
		return (this.a+this.b);
	}

	public double Soustraire ( double x, double y)
	{
		this.a = x;
		this.b = y;
		return (this.a - this.b);
	}
}

/*
* Classe MulDiv
*/
public class MulDiv
{
	public double a;
	public double b;


	public double Multiplier ( double x, double y)
	{
		this.a = x;
		this.b = y;
		return (this.a*this.b);
	}

	public double Diviser ( double x, double y)
	{
		this.a = x;
		this.b = y;
		return (this.a/this.b);
	}

}
```

Classes TESTCASES:

```
/*
* Classe de test de la classe AdSous
*/
import junit.framework.*;

public class AdSousTest extends TestCase
{
	private	AdSous		Calc;

	protected void setUp() throws Exception
	{
		super.setUp();
		this.Calc	= new AdSous();
	}

	protected void tearDown() throws Exception
	{
		super.tearDown();
		this.Calc = null;
	}

	public void testAdditionner() throws IOException
	{
		double R = Calc.Additionner ( 1, 3 );
		assertEquals( "Test Additionner-1+3: , (double) 4, R );
	}

	public void testSoustraire()
	{
		double R = Calc.Soustraire ( 2, 3);
		assertEquals( (double)-1, R );
	}
}


/*
* Classe de test de la classe MulDiv
*/
import junit.framework.TestCase;

public class MulDivTest extends TestCase
{
	private MulDiv		Calc;

	protected void setUp() throws Exception
	{
		super.setUp();
		this.Calc = new MulDiv();
	}

	protected void tearDown() throws Exception
	{
		super.tearDown();
		this.Calc = null;
```

```
	}

	public void testMultiplier()
	{
		double R = Calc.Multiplier ( 3, 5 );
		assertEquals( "Test Multiplier- 3*5: ",  (double) 15, R );
		}
	}

	public void testDiviser()
	{
		double R = Calc.Diviser ( 5, 2);
		assertEquals( "Test Diviser-5/2", (double) 2.5, R );
	}
}

/*
* Classe de test Test Suite (enchaîne les tests des deux classes
*/

import junit.framework.*;

public class ExecuterTests
{

  public static Test suite()
  {
    TestSuite suite = new TestSuite("Tous les tests");  //Crée une suite de tests

    suite.addTestSuite(AdSousTest.class); // Ajoute les tests de AdSous dans la suite
    suite.addTestSuite(MulDivTest.class); // Ajoute les tests de MulDiv dans la suite

    // et ainsi de suite si on veut enchaîner d'autres tests

    return suite;
  }
}
```

VI.2.9.SOLUTION POUR «FACTORISER» LES CAS DE TEST SUR DES SÉRIES DE VALEURS:

INTRODUCTION:

Certaines solutions sont proposées par JUNIT pour permettre la «factorisation» des cas de test. Elles permettent d'appeler automatiquement la fonction ASSERT d'un cas de test pour une série de valeurs d'entrée (en général contenues dans une table). Cependant, ces solutions sont soit compliquées, soit ne fonctionnement qu'avec certaines versions ou certains «plugins» de JUNIT.

Je propose ci-après une solution «maison» relativement simple à gérer. Cette méthode consiste à introduire une nouvelle classe JeuxEssai dont chacune des méthodes renvoie un tableau de valeurs. Chaque ligne du tableau comprend:

- Un jeu de valeurs d'entrées de la méthode à tester
- Les résultats attendus.

Exemple pour une fonction Multiplier:

Valeur du paramètre A	Valeur du paramère B	Valeur de A*B
3	8	25
7	-3	-21
5	0.8	4

Dans l'exemple qui suit, nous allons «factoriser» le test de la fonction Multiplier de MulDiv:

CLASSE JeuEssai:

Cette classe offre des méthodes qui permettent de récupérer des «jeux d'essai» (collections de valeurs de paramètres d'entrée et de valeurs de sorties correspondantes des fonctions à tester.

```
public class JeuxEssais
{
	public double[][] JeuMultiplier()
	{
		double T[][] =  {                       // Remarque: ces valeurs peuvent
                                //a   b  a*b        // être lues dans un fichier
                                { 1,  2,  3 },      // Ex: fichier csv issu d'un
                                { 2, -3, -6 },      // tableur.
                                { 3,  4, 12 }
                              };
		return ( T );
	}

	// On ajoute une méthode pour chaque jeu d'essai
}
```

NOUVELLE MÉTHODE testMultiplier:

```
public class MulDivTest extends TestCase
{
	private MulDiv	Calc;
	private JeuxEssais  JE;

	protected void setUp()
	{
		super.setUp();
		this.Calc = new MulDiv();
		this.JE   = new JeuxEssais();  // Instanciation de la classe JeuxEssai
	}

	protected void tearDown()
	{
		super.tearDown();
		this.Calc = null;
	}

	public void testMultiplier()
	{
		double TD[][]= this.JE.JeuMultiplier(); // Lecture du jeu d'essai
		for ( int i=0; i<TD.length; i++ )       // Factorisation du test pour
                                                // chaque ligne du jeu d'essai
		 {
		   double R = Calc.Multiplier ( TD[i][0], TD[i][1]);
		   assertEquals( "Test Multiplier-ligne "+(i+1)+":", (double)TD[i][2], R );
		}
	}

	public void testDiviser()
	{
		double R = Calc.Diviser ( 2, 3);
		assertEquals( "Test Diviser", (double)2/3, R );
	}
}
```