République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Mohamed Khider Biskra

Faculté des sciences exactes et des sciences de la nature et de la vie

Département d'Informatique

Niveau : 1ère année Master

Option : GLSD

Cours :

Technique de Test Logiciel (TTL)

Réalisé par :
Dr. Ouaar Hanane

Année universitaire 2022/2023

Contenu de la matière :

Chapitre I : Introduction à l'activité de test

Chapitre II : Technique de Test Logiciel (partie 1 et partie 2)

Chapitre III : Outils de Test

Chapitre VI : Initiation aux techniques de vérification formelle

Introduction générale

Le génie-logiciel est le domaine qui assiste dans toutes les phases d'un processus de développement d'un produit logiciel de haut degré de qualité avec un minimum d'erreur.

L'erreur est humaine, presque tous les programmes contiennent des erreurs. Les erreurs sont difficiles à identifier pour différentes raisons : la grande complexité des logiciels, l'invisibilité du système développé et l'absence de principe de continuité.

L'activité de test est essentielle au développement de logiciels de qualité. Lorsque les tests sont correctement réalisés et utilisés, ils permettent de découvrir des erreurs.

Les tests mettent en avant des défaillances du logiciel, c'est-à-dire des fonctionnements anormaux aux vues de ses spécifications.

La présente polycopie offre aux étudiants un volume de cours très souple et pertinent concernant l'activité de teste.

Chapitre 1 :

Introduction à l'activité de test

1. Définitions de base :

La présente section est consacrée à la définition des termes les plus proches de ce module :

1.1 Génie logiciel :

Le génie logiciel est un domaine des sciences de l'ingénieur dont l'objet d'étude est la conception, la fabrication et la maintenance des systèmes informatiques complexes.

1.2 Système Un système est un ensemble d'éléments interagissant entre eux suivant un certains nombres de principes et de règles dans le but de réaliser un objectif.

1.3 Logiciel : Un logiciel est un ensemble d'entités nécessaires au fonctionnement d'un processus de traitement automatique de l'information. Parmi ces entités, on trouve par exemple : Des programmes (en format code source ou exécutables). Des diagrammes. Des documentations d'utilisation. Des informations de configuration.

Le logiciel : est en général un sous-système d'un système englobant. Il peut interagir avec des clients, qui peuvent être : des opérateurs humains (utilisateurs, administrateurs), d'autres logiciels ; des contrôleurs matériels. Il réalise une spécification : son comportement vérifie un ensemble de critères qui régissent ses interactions avec son environnement.

1.4 Activités d'un cycle de vie

Le développement logiciel comprend un ensemble d'activités :

La gestion des exigences (définition des objectifs)
L'analyse des besoins et de faisabilité
La spécification
La conception générale
La conception détaillée
L'implantation (programmation)
Le teste unitaire (vérification)
Le teste d'intégration
La validation
L'intégration
Le déploiement
La maintenance
Elles fournissent différents produits logiciels (documents, modèles, code, …) .

2. Vérification & Validation : Deux aspects de la notion de qualité : Correction d'une phase ou de l'ensemble : VERIFICATION. Réponse à la question : faisons-nous le produit correctement ? Tests. Erreurs par rapport aux définitions précises établies lors des phases antérieures de développement

Conformité avec la définition : VALIDATION. Réponse à la question : faisons-nous le bon produit ? Contrôle en cours de réalisation, le plus souvent avec le client. Défauts par rapport aux besoins que le produit doit satisfaire Les spécifications fonctionnelles définissent les intentions. Elles sont créées lors de la phase d'analyse des besoins. La vérification du produit consiste à vérifier la conformité vis-à-vis de ces spécifications fonctionnelles. A ce stade on trouve des termes similaires : Revues, inspections, analyses, tests fonctionnels et tests structurel. La validation du produit consiste à vérifier par le donneur d'ordre la conformité vis-à-vis des besoins. Le plus souvent, tests fonctionnels en boîte noire. Théoriquement, la validation devrait

être plutôt faite par les utilisateurs, sans tenir compte du cahier des charges. En pratique, la validation s'appuie sur le cahier des charges pour créer des tests d'acceptation…

3. Principe de forte cohésion et de faible couplage

3.1 Couplage : est une métrique indiquant le niveau d'interaction entre deux ou plusieurs composants logiciels : fonctions, modules, objets ou applications. Deux composants sont dits couplés s'ils échangent de l'information. On parle de couplage fort ou couplage serré si les composants échangent beaucoup d'information. On parle de couplage faible, couplage léger ou couplage lâche si les composants échangent peu d'information et/ou de manière désynchronisée.

3.2 Cohésion peut être une métrique mesurant l'application des principes d'encapsulation des données et de masquage de l'information. Elle mesure également la cohésion sémantique des interfaces des modules et des classes. Application : On peut symboliser 1es concepts de cohésion et, de couplage dans le domaine des classes de la façon suivante : une classe a une forte cohésion si tous ses attributs sont, employés par ses méthodes et deux classes sont couplées si un changement dans l'une exige des changements dans l'autre. La cohésion forte indique que les méthodes de la classe collaborent pour accomplir une fonctionnalité, tandis que, le couplage faible indique que la classe est plutôt indépendante. Ainsi, il serait nécessaire de créer les applications en développant des composants avec : forte cohésion et faible couplage. En garantissant une forte cohésion et un faible couplage dans le développement d'une application, on gagne en qualité tout en diminuant la complexité. Bien que ce principe soit, simple à énoncer et à imaginer, il est difficile de le mettre en pratique.

4. Facteurs de qualité d'un logiciel

Les facteurs de qualités d'un logiciel (critères/métrique) sont classés selon trois points de vue.

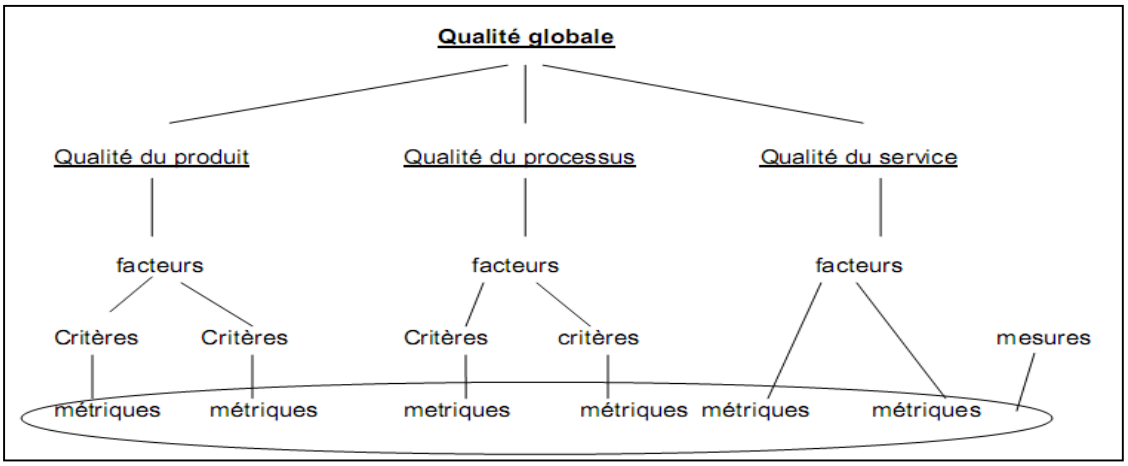


Figure 1 : Structure des facteurs de qualités

Un facteur est une caractéristique du logiciel, du processus ou du service contribuant à sa qualité telle qu'elle est ressentie par l'utilisateur. **Un critère** est un attribut du logiciel par l'intermédiaire duquel un facteur peut être obtenu. C'est également une caractéristique du logiciel sur laquelle le développeur peut agir. (Par exemple, sa simplicité). **Une métrique** est la mesure d'une propriété d'un critère. (Par exemple, la taille d'un module pour le critère "Simplicité").

4.1 Point de vue fonctionnel :

Pertinence : C'est la capacité de répondre au problème de l'entreprise.

Adéquation : C'est l'adéquation du logiciel à l'organisation et aux procédures de l'entreprise.

Généralité : C'est l'aptitude de la solution à résoudre des problèmes de portée plus large que le contexte particulier du projet.

4.2 Point de vue utilisation :

Maniabilité :Aptitude du logiciel à être convivial et simple d'emploi.

Fiabilité : Aptitude du logiciel à accomplir sans défaillance l'ensemble de fonctions spécifiées dans un document de référence pour une durée d'utilisation donnée.

Efficience : Aptitude du logiciel à minimiser l'utilisation des ressources disponibles.

Confidentialité : Aptitude du logiciel à être protégé contre tout accès par des personnes non autorisées, aussi bien en que hors exploitation.

Couplabilité (ou interopérabilité) : Aptitude du logiciel à communiquer ou interagir avec d'autres système.

4.3 Point de vue « maintenance » :

Maintenabilité : Aptitude avec laquelle on peut corriger les erreurs résiduelles et à évoluer facilement.

Portabilité : Aptitude à transférer le logiciel dans un autre environnement.

5. Critères de qualité d'un logiciel

- **La maniabilité :** comporte trois critères **;** La communicabilité est la capacité du logiciel de permettre un dialogue aisé entre l'humain et la machine. L'exploitabilité est la facilité à mettre en œuvre et à utiliser le logiciel (sauvegardes, restauration, reprise en cas d'erreur, etc.). La facilité d'apprentissage est la capacité du logiciel à être utilisé rapidement par un utilisateur final.
- **La fiabilité** se décline aussi en trois critères : La complexité du code. La tolérance aux fautes est la possibilité de limiter les effets d'une perturbation interne ou externe, sur le logiciel.
- **L'auditabilité** est la capacité du logiciel de permettre de retrouver rapidement et aisément la trace d'une opération.
- **L'efficience :** comporte la place mémoire, la vitesse d'accès aux périphériques, les temps de réponse, etc.
- **La confidentialité :** La protection du code et des données est la limitation, en exploitation ou non, des accès à des personnes autorisées. La mémorisation des accès est l'historisation des accès aux données et aux fonctions.
- **La couplabilité** (ou interopérabilité) : La standardisation des données est la compatibilité des données avec des standards de représentation. La standardisation des interfaces.
- **La maintenabilité :** La lisibilité est la capacité d'un logiciel et de sa documentation a depouvoir être lus par d'autres personnes que leur auteur. La modularité représente l'indépendance des composants. La traçabilité permet de retrouver l'origine des éléments.

  L'adaptabilité est l'aptitude du logiciel à évoluer aisément.
- **La portabilité :** La banalité d'emploi traduit l'indépendance de la fonction par rapport à l'application. L'indépendance est le degré de liberté du logiciel par rapport à l'environnement. La qualité de la documentation sur le fond et la forme.

6. **Définitions de l'activité de test** :
   - Le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus.
   - Le test est une technique de contrôle consistant à s'assurer, au moyen de son exécution, que le comportement d'un programme est conforme à des données préétablies.
   - Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts.

7. **Tester le document (manuel) d'un logiciel :** son but est de :
   - Minimisation des problèmes d'interprétation : Point de vue indépendant du rédacteur
   - Vérification : Forme : respect des normes, précision, non ambiguïté. Fond : cohérence et complétude. Testabilité et traçabilité.
   - Validation : Mauvaises interprétations des documents de référence. Critères de qualité mal appliqués. Hypothèses erronées. Coût se calcule comme : 5 à 10 p (page)/h (heur) pour le cahier des charges. 20 à 50 LOC (Ling Of Code)/h pour le code.

8. **La chaîne de l'erreur**
   - **Faute** : Une faute dans un logiciel ou un système peut être vu comme un élément ou un évènement ayant entraîné une erreur dans celui-ci. Par exemple, une faute dans un logiciel est le plus souvent d'origine humaine : erreur d'interprétation d'une exigence, oubli d'une condition dans une expression booléenne, etc. Cependant, une faute peut exister sans pour autant perturber le fonctionnement du logiciel, dans ce cas elle est dite dormante ou passive. Dans le cas contraire, si son activation produit une erreur, elle est dite active.
   - **Erreur :** Une erreur dans un système est un état non spécifié, atteint par celui-ci. Elle pourrait par exemple être une "mauvaise" valeur d'un paramètre d'une fonction, mauvaise initialisation d'une variable due à une faute, etc. Une erreur qui se propage dans le logiciel peut entraîner une défaillance de celui-ci.
   - **Défaillance :** Une défaillance d'un système est un état dans lequel le service délivré par celui-ci n'est pas conforme au service attendu. En d'autres termes, une différence est observée entre le comportement fourni par le système et le comportement attendu (vis à vis des spécifications). Une défaillance peut avoir des conséquences plus ou moins graves dépendant de sa criticité. En général, on les classe en fonction de leur niveau de criticité. Par exemple, dans le cas de l'automobile, les défaillances sont souvent classées selon les quatre niveaux de criticité suivants : mineure, majeure, sérieuse et fatale.

Exemple : Supposons que le programmeur au lieu d'écrire l'instruction "x = a + b;" ait écrit l'instruction "x = a- b;" voir le code encadré. Cette dernière instruction est une **faute**. Si lors des différentes évaluations de la condition "(a == b)", a et b ne sont pas égaux, alors il n'y aura peut-être pas **d'erreur** malgré la présence de la faute. Sinon si a et b sont égaux et que la valeur de k n'est pas égale à la valeur 1, il n'y aura pas de défaillance malgré la présence de **l'erreur** (activation de la faute "x = a - b"). Par contre si la valeur de k vaut 1, dans ce cas on a une division par 0 et il y a **défaillance**.

```
Int a, b, x, y, k ;
……
if (a==b) then  x=a-b ;
end if ;
if (k==1) then  y=y/x ;
end if
```

9. Définitions de quelques termes autour de test

- **Exigence :**Une condition ou capacité requise par un utilisateur pour résoudre un problème ou atteindre un objectif, qui doit être tenu ou possédé par un système ou composant pour satisfaire à un contrat, standard, spécification ou autre document imposé formellement.
- **Spécification :**Un document qui spécifie, idéalement de façon complète, précise et vérifiable, les exigences, conceptions, comportements et autres caractéristiques d'un composant ou système, et souvent, les procédures pour déterminer si ces stipulations ont été satisfaites.
- **Entrées de test :** Données reçues d'une source externe par l'objet de test pendant son exécution. Les sources externes peuvent être matérielles, logicielles ou humaines.
- **Donnée de test :** Donnée existante (par exemple, dans une base de données) avant qu'un test ne soit exécuté et qui affecte ou est affectée par le composant ou système en test.
- **Cas de test :** Un ensemble de valeurs d'entrée, de pré- conditions d'exécution, de résultats attendus et de post-conditions d'exécution, développées pour un objectif ou une condition de test particulier, tel qu'exécuter un chemin particulier d'un programme ou vérifier le respect d'une exigence spécifique. Un cas de test est un ensemble composé de trois objets : Un état (ou contexte) de départ ; Un état (ou contexte) d'arrivée ; Un oracle, c'est à dire un outil qui va prédire l'état d'arrivée en fonction de l'état de départ et comparer le résultat théorique et le résultat pratique. Un cas de test peut donc s'appliquer à plusieurs méthodes, par exemple plusieurs classes implémentant la même interface.
- **Test :** Un ensemble d'un ou plusieurs cas de tests. **Suite de test :** Ensemble de plusieurs cas de tests pour un composant ou système à tester, dont les post-conditions d'un test sont souvent utilisées comme pré conditions du test.
- **Stratégie de test :** Document de haut niveau définissant, pour un programme, les niveaux de tests à exécuter et les tests dans chacun de ces niveaux (pour un ou plusieurs projets).

10. Processus de Test logiciel : il est constitué d'étapes suivantes :

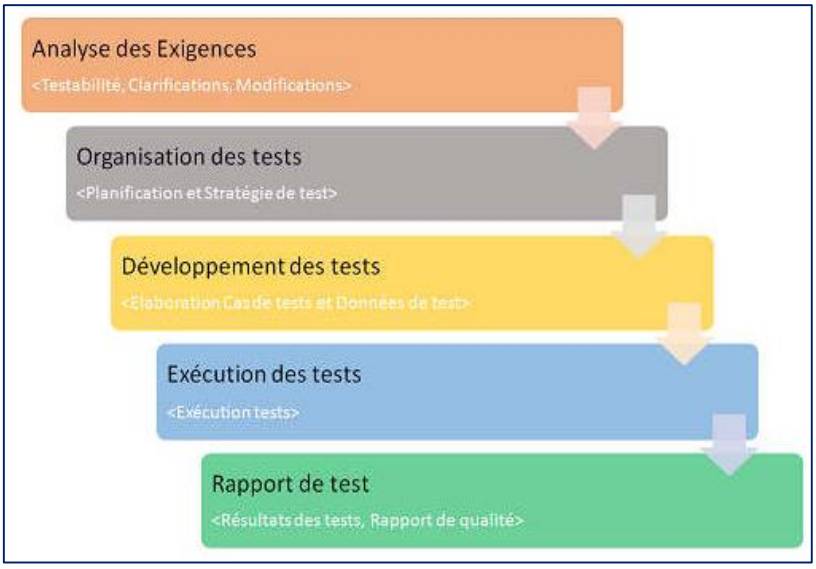


Figure 2 : Structure des processus de test

- **Analyse des exigences :** Dans cette étape les différentes exigences sont analysées afin d'évaluer leur testabilité et d'apporter si nécessaire des clarifications et des modifications.
- **Organisation des tests :** Cette étape comprend la définition de la stratégie de test et la planification des activités de test. Par exemple, les différentes approches à utiliser, le temps à allouer aux activités de test, les outils à utiliser, les livrables, sont spécifiés dans cette partie.
- **Développement ou conception des tests :** Cette étape consiste au développement des différents types de tests en tenant compte de l'étape précédente. Dans un premier temps, les tests sont définis de façon générique avec des résultats attendus (cas de test). Ensuite ces tests abstraits seront utilisés pour produire des données de test. Cette étape est cruciale dans le processus de test car celle-ci joue un rôle très important dans la qualité des tests finaux. C'est dans cette étape que les différents tests sont conçus pour vérifier et valider les différentes parties du logiciel Développement ou conception des tests. Cette étape consiste au développement des différents types de tests en tenant compte de l'étape précédente. Dans un premier temps, les tests sont définis de façon générique avec des résultats attendus (cas de test). Ensuite ces tests abstraits seront utilisés pour produire des données de test. Cette étape est cruciale dans le processus de test car celle-ci joue un rôle très important dans la qualité des tests finaux. C'est dans cette étape que les différents tests sont conçus pour vérifier et valider les différentes parties du logiciel.
- **Exécution des tests :** Dans cette étape, les différentes données de test sont exécutées pour produire des résultats d'exécution.
- **Rapport de test :** Dans cette étape les différents résultats d'exécution des tests sont analysés afin d'établir un rapport de test. Ainsi, les différentes anomalies (différence entre le résultat d'exécution du test avec le résultat attendu) liées à l'exécution des tests, la qualité des tests et bien d'autres informations seront reportées dans cette partie.

Université de Mohamed Khider – Biskra –

Département d'informatique

Niveau : 1ère Master (GLSD) **Module : TTL**

Année : 2023

TD N°1 :

Exercice 1 :

Proposer un programme P développé sous Java selon votre choix. Rendre un compte rendu contenant les éléments suivants :

- Elaborer son diagramme de classe.
- Elaborer d'autres diagrammes au choix, séquence, use cas et activités.
- Vérifier toutes les propriétés de ce code avec son spécification.
- Corriger le code si nécessaire.
- Proposez un modèle de validation de ce code avec le user
- Corriger le code si nécessaire aussi.
- Vérifiez le degré de couplage par rapport au diagramme de classe/objet
- Vérifiez le degré de cohérence par rapport au diagramme de séquence
- Présenter le code final
- Proposez ces facteurs de qualité de présent code
- Pour chaque facteur proposez ces critères de qualité les plus adéquates

Exercice 2 :

Trouvez les équations nécessaire pour calculer le (coût + délai) de teste de chaque phase d'un processus de développement logiciel.

Chapitre 2:

Techniques de Test Logiciel

-Partie N°1-

1. **Classification des tests :** Il existe différentes façons de classifier les tests. Il est possible de les regrouper selon : leur mode d'exécution, leurs modalités, leurs méthodes, leurs niveaux.

1.1 Le mode d'exécution

- **Le test Manuel** Les tests sont exécutés par le testeur. Il saisit les données en entrée, vérifie les traitements et compare les résultats obtenus avec ceux attendus. Ces tests sont fastidieux et offrent une plus grande possibilité d'erreurs humaines. Ces tests sont très vite ingérables dans le cas d'applications de grandes tailles.
- **Le test Automatique :** L'automatisation des tests est «l'utilisation de **logiciels** pour exécuter ou supporter des activités de tests, par exemple : gestion des tests, conception des tests, exécution des tests ou vérification des résultats». JUnit par exemple dans le monde Java.

1.2 Les modalités de test

- **Statiques** : Les tests sont réalisés «par l'humain» (testeur), sans machine, par lecture du code dans le but de trouver des erreurs. Il peut s'agir : de l'inspection ou d'une revue de code ; d'un travail de collaboration lors d'une réunion : Le programmeur, le concepteur, un programmeur expérimenté, un testeur expérimenté, un modérateur (régulateur)…=> Portent sur des documents (plutôt des programmes), sans exécuter le logiciel.

- **Avantages :** Contrôle systématique valable pour toute exécution, applicables à tout document.
- **Inconvénients** Ne portent pas forcément sur le code réel. - Ne sont pas en situation réelle (interaction, environnement). Vérifications sommaires, sauf pour les preuves. Ces preuves nécessitent des spécifications formelles et complètes**,** donc difficiles.

- **Dynamiques** : On exécute le système de manière à tester l'ensemble des caractéristiques. Chaque résultat est comparé aux résultats attendus. => Nécessitent une exécution du logiciel, une parmi des multitudes d'autres possibles

- **Avantages :** Vérification avec des conditions proches de la réalité. Plus à la portée du commun des programmeurs.
- **Inconvénients :** Il faut provoquer des expériences, donc écrire du code et construire des données d'essais. Un test qui réussit ne démontre pas qu'il n'y a pas d'erreurs. =>Les techniques statiques et dynamiques sont donc complémentaires.

1.3 Les méthodes de test : Il existe trois méthodes de test **:** blanche, noire et grise.

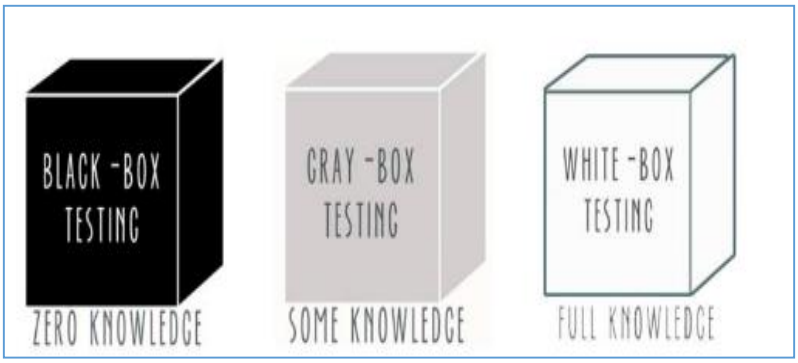
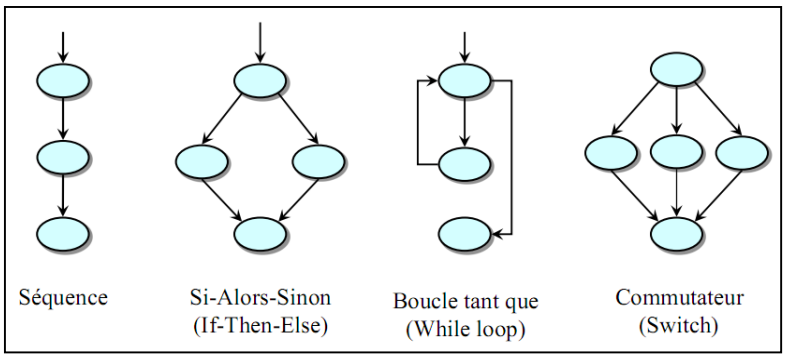


Figure 1 : méthodes de test

1.3.1 Structurels (Boîte blanche, Glass box) : Les tests structurels reposent sur des analyses du code source. Il est possible d'analyser la structure du programme. Le principe est de : Accès au **code** pour déduire des tests à faire, de manière plus complète. Test de couverture : passage par tous les blocs d'instructions solidaires (GFC : Graphe de flot de contrôle. Techniques d'analyse dynamique, en effectuant l'instrumentation du code (logiciel de test).

- **Graphe de flot de contrôle :** c'est un Graphe orienté. Les nœuds sont des blocs d'instructions séquentiels. Les arêtes sont des transferts de contrôle. Les arêtes peuvent être étiquetées avec un attribut représentant la condition du transfert de contrôle. Plusieurs conventions existent pour les modèles de graphes de flot avec des différences subtiles




```
read(x);
read(y);
while x != y loop
   if x>y then
        x := x - y;
   else
        y := y - x;
   end if;
end loop;
gcd := x;
```

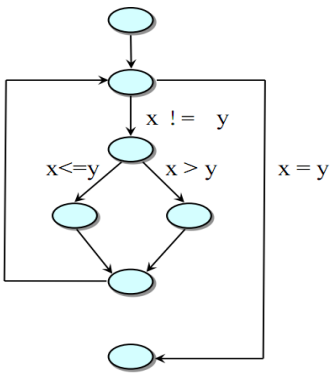


Figure 2 : graphe de flot de control

a- **Tests des chemins d'exécution et de couverture :** A partir du graphe de flux de contrôle, on vérifie que l'on passe par tous les blocs d'instructions solidaires. Plusieurs niveaux de contrôle sont envisageables :

- **Tests des chemins d'exécution :** Toutes les combinaisons possibles de passage de l'entrée à la sortie de la routine. => Trop coûteux
- **Tests des chemins indépendants :** Contiennent au moins un arc n'appartenant pas aux autres chemins. =>Moins coûteux
- **Tests de couverture :** S'assure qu'on passe au moins une fois dans chaque bloc

b- **Test de couverture de chemins** : Dériver une mesure de la complexité logique. L'utiliser pour définir un ensemble de base des chemins d'exécution. D'où calcul de la complexité cyclomatique. **V(G)** fournit une borne supérieure des tests qui doivent être exécutés pour garantir la couverture de toutes les instructions des programmes. Le nombre de composants connectés : => sous graphe

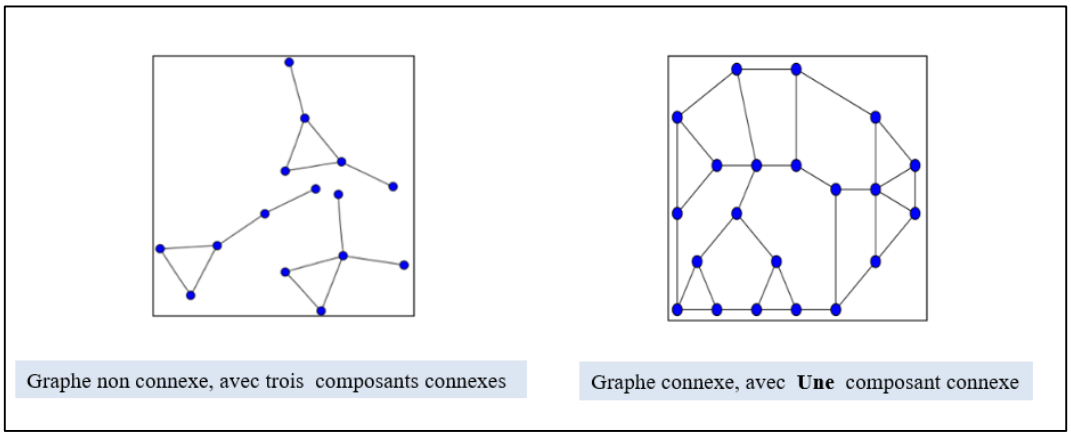


Figure 3 : composants connectés

Le nombre cyclomatique, la complexité cyclomatique ou la mesure de McCabe en 1976 est un outil de métrologie logicielle développé par Thomas McCabe en 1976 pour mesurer la complexité d'un programme informatique. Cette mesure reflète le nombre de décisions d'un algorithme en comptabilisant le nombre de « chemins » linéairement indépendants au travers d'un programme représenté sous la forme d'un graphe.

La complexité cyclomatique V(G) définit le nombre de chemins indépendants dans l'ensemble de base. L'ensemble de couverture des chemins est alors l'ensemble des chemins qui exécuteront toutes les instructions et toutes les conditions au moins une fois dans un programme. Les tests de couverture doivent être appliqués aux modules critiques. De nombreuses études industrielles ont montré que plus V(G) est grand, plus la probabilité d'erreur est importante.

Exemple 1 : supposant l'algorithme de la recherche dichotomique d'un élément dans un tableau : Dans ce cas, V (G) = 11 -9 + 2*1 = 4 Comme **V(G) = 4➔ il y a 4 chemins**
Chemin 1: 1,2,3,6,7,8
Chemin 2: 1,2,3,5,7,8
Chemin 3: 1,2,4,7,8
Chemin 4: 1,2,4,7,2,4…7,8

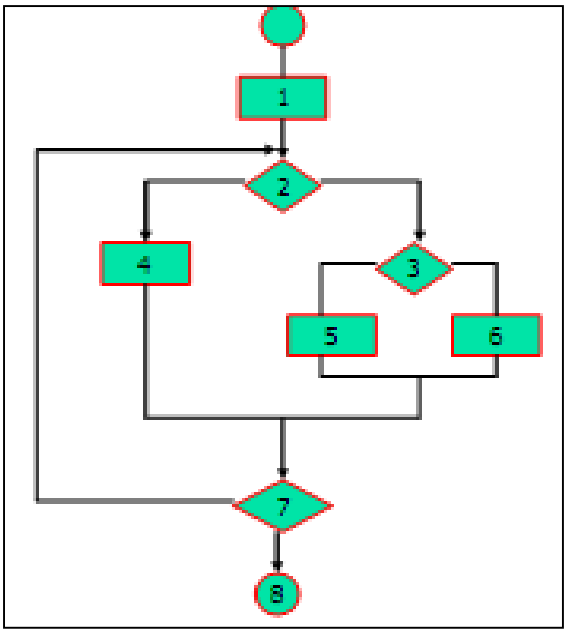


Figure 4 : exemple d'un GFC

c- **Affinage des jeux de tests :** L'examen du code montre qu'à chaque itération, le tableau est partitionné en 3 parties : Partie dont les indices sont inférieurs au milieu. Élément du milieu. Partie dont les sont supérieurs au milieu**.** On peut ainsi ajouter les tests suivants : Clé présente au centre du tableau. Clé présente juste avant le centre du tableau. Clé présente juste après le centre du tableau**.** Solution Assez empirique !

Exemple 2 : le calcule de PGCD

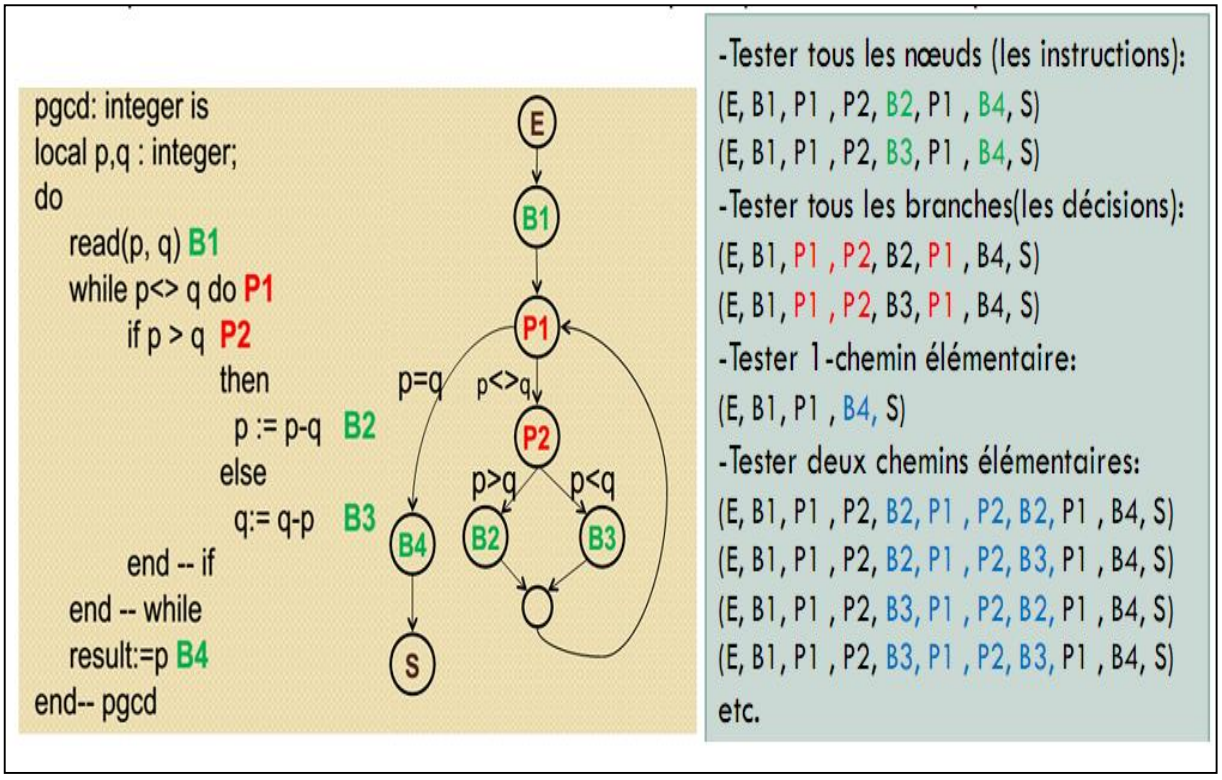


Figure 5 : GFC de PGCD

d- Avantages de la méthode boite blanche :

- **Anticipation :** effectuer ces tests au cours du développement d'un programme permet de repérer des points bloquants qui pourraient se transformer en erreurs ou **problèmes** dans le futur (par exemple lors d'une montée en version, ou même lors de l'intégration du composant testé dans le système principal).
- **Optimisation :** étant donné qu'il travaille sur le code, le testeur peut également profiter de son accès pour optimiser le code, pour apporter de meilleures performances au système étudié (sans parler de sécurité…).
- **Exhaustivité :** étant donné que le testeur travaille sur le code, il est possible de vérifier intégralement ce dernier. C'est le type de test qui permet, s'il est bien fait, de tester l'ensemble du système, sans rien laisser passer. Il permet de repérer des bugs et vulnérabilités cachées intentionnellement.

e- Inconvénients de la méthode boite blanche :

- **Complexité :** ces tests nécessitent des compétences en programmation, et une connaissance accrue du système étudié.
- **Durée** : de par la longueur du code source étudié, ces tests peuvent être très longs.
- **Industrialisation :** pour réaliser des tests en « boîte blanche », il est nécessaire de se munir d'outils tels que des analyseurs de code, des débogueurs… Cela peut avoir un impact négatif sur les performances du système, voire même impacter les résultats.
- **Cadrage :** il peut être très compliqué de cadrer le projet. Le code source d'un programme est souvent très long, il peut donc être difficile de déterminer ce qui est testé, ce qui peut être mis de côté… En effet, il n'est pas toujours réaliste de tout tester, ce qui prendrait trop de temps. Il est également possible que le testeur ne se rende pas compte qu'une fonctionnalité prévue dans le programme n'y a pas été intégrée. Il n'est donc pas dans le scope du testeur de vérifier si tout est là : il ne fait que tester ce qui est effectivement présent dans le code.
- **Intrusion :** cette méthode est très intrusive. Il peut en effet être risqué de laisser son code à la vue d'une personne externe à son entreprise : il y a des risques de casse, de vol, … Choisissez donc toujours des testeurs professionnels !

1.3.2 Les méthodes de test : Fonctionnels (Boîte noire)

Les tests en « boite noire » consistent à examiner uniquement les fonctionnalités d'une application, c'est-à-dire si elle fait ce qu'elle est censée faire, peu importe comment elle le fait. Sa structure et son fonctionnement interne ne sont pas étudiés. Le testeur doit donc savoir quel est le rôle du système et de ses fonctionnalités, mais ignore ses mécanismes internes. Il a un profil uniquement « utilisateur ».

Cette méthode sert à vérifier, après la finalisation d'un projet, si un logiciel ou une application fonctionne bien et sert efficacement ses utilisateurs. En général, les testeurs sont à la recherche de fonctions incorrectes ou manquantes, d'erreurs d'interface, de performance, d'initialisation et de fin de programme, ou bien encore d'erreurs dans les structures de données ou d'accès aux bases de données externes.

Les tests fonctionnels ou boîte opaque reposent sur une spécification du programme. Le code source du programme n'est pas utilisé. Les tests fonctionnels permettent d'écrire les tests bien avant le «codage ». Il est parfois utile de combiner ces deux méthodes.

Le test de la boîte noire est également connu sous différents noms : Tests basés sur les spécifications. Test comportemental. Tests basés sur les données. Test d'entrée-sortie. Test black box (traduction en anglais). Elle applique dans certains domaines : transistor, algorithmes et réseau comme internet. La méthode de test fonctionnel repose sur six principes :

-Partitionnement en classes d'équivalence. -Analyse de la valeur limite. -Test catégorie-partition. -Tables de décision. -Graphes de cause-à-effet. -Fonctions logique

a- Partitionnement en classes d'équivalence : nous aimerions obtenir un sens de test complet et nous souhaiterions éviter la redondance. Classe d'équivalence : partage de l'ensemble des données. Ensemble des données entier est couvert : complet. Classes séparées : éviter la redondance. Cas de tests : un élément de chacune des classes d'équivalence. Mais les classes d'équivalence doivent être choisies prudemment … Estimer le comportement probable du système à tester.

- **Test par classes d'équivalence faible :** choisir une forme de variable d'entrée pour chacune des classes d'équivalences : **max (|A|, |B|, |C|) cas de test**
- **Test par classes d'équivalence fort :** basé sur le produit Cartésien des sous-ensembles de partition (A, B et C), les interactions de classes est calculé comme :

|A| x |B| x |C| cas de test

Exemple : NextDate est une fonction avec trois variables : MOIS, JOUR, ANNEE. Elle retourne la date du jour après la donnée date. Limites : 1812-2012. Sommaire du traitement :

- si ce n'est pas le dernier jour du mois, la dernière fonction date augmente simplement la valeur **JOUR**.

- À la fin du mois, le jour suivant est 1 et la valeur **MOIS** est augmentée.

- À la fin d'une année, les deux valeurs JOUR et **MOIS** sont remises à 1, et la valeur **ANNEE** est augmentée.

- Finalement, le problème de l'année **bissextile** rend intéressant la détermination du dernier jour d'un mois.

- NextDate: 36 cas de tests possibles!!!

Cas ID	Mois	Jour	An	Sortie anticipée
SE1	6	14	1900	6/15/1900
SE2	6	14	1912	6/15/1912
SE3	6	14	1913	6/15/1913
SE4	6	29	1900	6/30/1900
SE5	6	29	1912	6/30/1912
SE6	6	29	1913	6/30/1913
SE7	6	30	1900	7/1/1900
SE8	6	30	1912	7/1/1912
SE9	6	30	1913	7/1/1913
SE10	6	31	1900	ERREUR
SE11	6	31	1912	ERREUR
SE12	6	31	1913	ERREUR
SE13	7	14	1900	7/15/1900
SE14	7	14	1912	7/15/1912
SE15	7	14	1913	7/15/1913
SE16	7	29	1900	7/30/1900
SE17	7	29	1912	7/30/1912
SE18	7	29	1913	7/30/1913
SE19	7	30	1900	7/31/1900
SE20	7	30	1912	7/31/1912
SE21	7	30	1913	7/31/1913
SE22	7	31	1900	8/1/1900
SE23	7	31	1912	8/1/1912
SE24	7	31	1913	8/1/1913
SE25	2	14	1900	2/15/1900
SE26	2	14	1912	2/15/1912
SE27	2	14	1913	2/15/1913
SE28	2	29	1900	ERREUR
SE29	2	29	1912	3/1/1912
SE30	2	29	1913	ERREUR
SE31	2	30	1900	ERREUR
SE32	2	30	1912	ERREUR
SE33	2	30	1913	ERREUR
SE34	2	31	1900	ERREUR
SE35	2	31	1912	ERREUR
SE36	2	31	1913	ERREUR

Figure 6 : cas de tests possible

- **Construction de class es d'équivalence**

M1 = { MOIS: MOIS à 30 jours} M2 = { MOIS: MOIS à 31 jours}

M3 = { MOIS: MOIS est Février}

D1 = {JOUR: 1<= JOUR <= 28}

D2 = {JOUR: JOUR = 29} D3 = {JOUR: JOUR = 30} D4 = {JOUR: JOUR = 31}

Y1 = {ANNEE: ANNEE = 1900}

Y2 = {ANNEE: 1812 <= ANNEE <= 2012 et (ANNEE != 1900) et (ANNEE mod 4 = 0)}
Y3 = {ANNEE: (1812 <= ANNEE <= 2012 et ANNEE mod 4 != 0)}

b- Analyse de la valeur limite : Nous avons partitionné les domaines de données en classes appropriées, sous la supposition que le comportement du programme est "similaire". Quelques erreurs typiques de programmation se passent à la limite entre les différentes classes. C'est sur quoi le test de valeur limite se concentre. Plus simple mais complémentaire aux techniques précédentes.

Exemple : Supposons une fonction F, avec deux variables x1 et x2. Possiblement sans état Limites : a <= x1 <= b, c <= x2 <= d. Dans quelques langages de programmation, un fort typage permet la spécification de tels intervalles. Se concentre sur les bornes de l'espace d'entée

pour identifier les cas de test. Le raisonnement est que les erreurs tendent à se produire extrêmement près des valeurs des variables d'entée, ceci est supporté par quelques études.

- **Idées fondamentales :** Les valeurs des variables d'entrée à leur minimum, juste au-dessus du minimum, à une valeur nominale, juste en dessous de leur maximum, et à leur maximum. Convention : min, min+, nom, max-, max. Garde les valeurs de tout sauf une seule variable à leurs valeurs nominales, laissant une seule variable suppose sa valeur extrême.

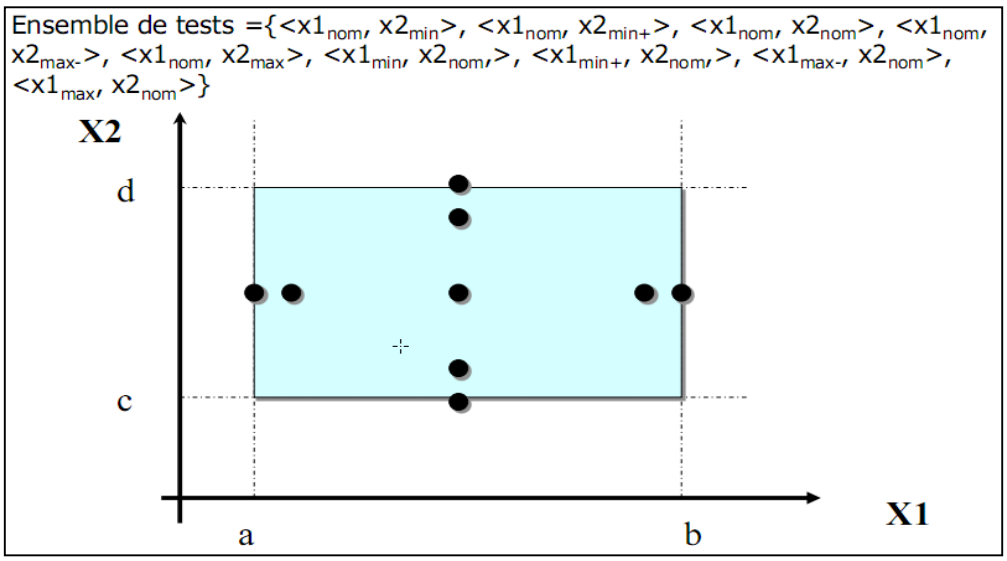


Figure 7 : position des valeurs limites

- **Cas général et limitations** : Une fonction avec n variables nécessitera 4n + 1 cas de tests. Fonctionne bien avec les variables qui représentent des quantités physiques limitées. Sans considération de la nature de la fonction et le sens des variables. Une technique rudimentaire (simple, claire) qui est adéquate au **test de robustesse**.

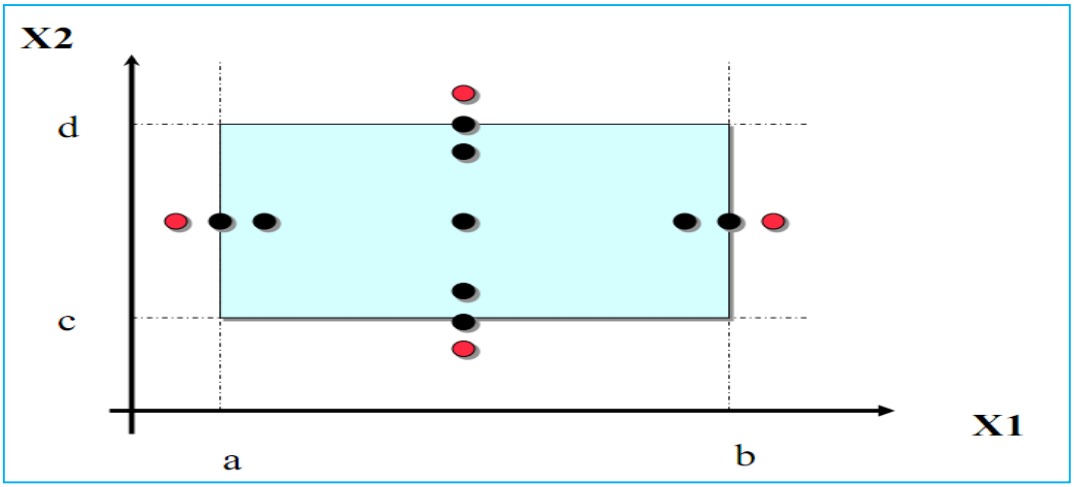


Figure 8 : position de test de robustesse

c- Test catégorie- partition :

Étapes I : Le système est divisé en "fonctions" individuelles qui peuvent être testées individuellement. La méthode identifie les paramètres de chaque "fonction" et pour chaque paramètre, identifie des catégories distinctes. En plus des paramètres, les caractéristiques environnementales, sous lesquelles la fonction opère (caractéristiques de l'état du système), peuvent aussi être considérées. Les catégories sont des propriétés majeures ou caractéristiques. De plus, les catégories sont subdivisées en choix de la même manière que le partitionnement par **classe d'équivalence** qui est appliqué ("valeurs" possibles).

- Fonction : tri d'une matrice
- Caractéristiques :
  - Longueur de la matrice (Len)
  - Type d'éléments
  - Valeur max
  - Valeur min
  - Position de la valeur max (max pos)
  - Position de la valeur min
- Choix pour Max pos : {1.2.Len-1, Len}

Étapes II : Les contraintes opérant entre les choix sont alors identifiées, i.e., comment l'occurrence d'un choix peut affecter l'existence d'un autre. Dans un exemple de tri de tableau, si Len = 0, alors le reste n'a pas d'importance. Les trames de test "test frames" sont générés, ce qui consiste en combinaisons admissibles de choix dans les catégories (test de spécifications). Les trames de test sont alors converties en données de test.
Contraintes : Les propriétés, les sélecteurs associés avec les choix.

Catégorie A :
Choix A1 [propriété X, Y, Z],
Choix A2.

Catégorie B :
Choix B1,
Choix B2 [si X et Z].

Annotation spéciale : [Erreur], [Simple]

- **Spécification :** Le programme demandant à l'usager de taper un nombre entier positif dans un champ de 1 à 20 et une chaîne de caractères de cette longueur. Le programme, alors, demande de taper un caractère et retourne la position dans la chaîne où le caractère a été trouvé en premier ou un message indiquant que ce caractère n'était pas présent dans la chaîne. L'usager a l'option de chercher d'autres caractères**.**
- **Paramètres et catégories** Trois paramètres : nombre entier x (longueur), la chaîne a et le caractère c.

* Pour x, les catégories sont dans la séquence "in- range" (1-20) ou hors de la séquence "out-of-range".

* Les catégories pour a : minimal, maximal, longueur intermédiaire.

* Les catégories pour c : le caractère apparaît au : commencement, au milieu, à la fin de la chaîne, ou ne se produit pas dans la chaîne.

➢ **Choix des cas possibles** :

* Nombre entier x, hors de la séquence (out-of-range) : 0, 21.

* Nombre entier x, dans la séquence (in-range) : 1, 2-19, 20.

* Chaîne a : 1, 2-19, 20.

* Caractère c : premier, milieu, dernier, ne se produit pas.

* Remarque : parfois, il n'y a qu'un choix dans la catégorie.

➢ **Spécifications du test formel :** c'est l'étude de déférentes plages de données possible

```
x:
x1)  0                                  [erreur]
x2)  1                                  [propriété chaineok, longueur1]
x3)  2-19                               [propriété chaineok, midlongueur]
x4)  20                                 [propriété chaineok, longueur20]
x5)  21                                 [erreur]
a:
a1)  longueur 1                         [si chaineok et longueur1]
a2)  longueur 2-19                      [si chaineok et midlongueur]
a3)  longueur 20                        [si chaineok et longueur20]
c:
c1)  à la premiere  first position dans la  chaine [si chaineok]
c2)  à la derniere  position dans la  chaine [si chaineok et not longueur1]
c3)  au milieu de la chaine             [si chaineok et not longueur1]
c4)  pas dans la  chaine                 [si chaineok]
```

Figure 9 : Spécifications du test formel

➢ **Trames de test et de cas de test possible :** c'est la réalisation de l'étude précédente

```
x 1          x = 0
x 2a1c1      x = 1, a = 'A' , c = 'A'
x 2a1c4      x = 1, a = 'A' , c = 'B'
x 3a2c1      x = 7, a = 'ABCDEFG' , c = 'A'
x 3a2c2      x = 7, a = 'ABCDEFG' , c = 'G'
x 3a2c3      x = 7, a = 'ABCDEFG' , c = 'D'
x 3a2c4      x = 7, a = 'ABCDEFG' , c = 'X'
x 4a3c1      x = 20, a = 'ABCDEFGHIJKLMNOPQRST' , c = 'A'
x 4a3c2      x = 20, a = 'ABCDEFGHIJKLMNOPQRST' , c = 'T'
x 4a3c3      x = 20, a = 'ABCDEFGHIJKLMNOPQRST' , c = 'J'
x 4a3c4      x = 20, a = 'ABCDEFGHIJKLMNOPQRST' , c = 'X'
x 5          x = 21
```

Figure 10 : Trames de test

- **Synthés de test catégorie-partition :** L'identification des paramètres et des conditions d'environnement, et des catégories dépend fortement de l'expérience du testeur. Rend le test des décisions explicite (e.g., contraintes), prêt pour l'évaluation. Combine l'analyse des valeurs limites, le test de robustesse et le partitionnement par classes d'équivalences. Un fois que la première étape est complétée, la technique est simple et peut être automatisée. La technique permet la réduction de cas de test ce qui le rend utile pour le test pratique.

d- Tables de décision (Table de vérité) : son principe est de :

* Aide à exprimer les spécifications de test directement dans une forme utilisable.

* Facile à comprendre et supporte la dérivation systématique des tests.

* Supporte la génération des cas de tests automatiques ou manuels.

* Une réponse particulière ou un sous-ensemble de réponse doit être choisie en évaluant plusieurs conditions reliées.

* Idéal pour décrire les situations dans lesquelles un nombre de combinaisons d'actions sont prises selon des ensembles variables de conditions, e.g. systèmes de contrôle.

- **Structure d'une table de décision**

***** La section condition liste les conditions et les combinaisons correspondantes

* La condition exprime l'association entre les variables de décision.

* La section action liste les réponses à être produites quand les combinaisons correspondantes des conditions sont vraies.

* Limitations : les actions résultantes sont déterminées par les valeurs courantes des variables de décision !

* Les actions sont indépendantes de l'ordre de des entrées et de l'ordre dans lequel les conditions sont évaluées.

* Les actions peuvent apparaître plus d'une fois mais chaque combinaison de conditions est unique.

Exemple : Soit a, b, c trois longueurs (entrées) de côté triangulaire :

Son tables de décision ou Table de vérité de type condition /action est présentée comme suite :

conditions											
c1: a < b + c?	F	T	T	T	T	T	T	T	T	T	T
c2: b < a + c?	-	F	T	T	T	T	T	T	T	T	T
c3: c < a + b?	-	-	F	T	T	T	T	T	T	T	T
c4: a = b?	-	-	-	T	T	T	T	F	F	F	F
c5: a = c?	-	-	-	T	T	F	F	T	T	F	F
c6: b = c?	-	-	-	T	F	T	F	T	F	T	F
a1: Pas un triangle	X	X	X								
a2: Scalène											X
a3: Isocèles							X		X	X	
a4: Équilatéral				X							
a5: Impossible					X	X		X			

Figure 11 : Table de vérité

- **Tables de décision (Table de vérité) :** c'est l'ensemble de cas de test d'après figure 11

Cas ID	a	b	c	Sortie anticipée
TC1	4	1	2	Pas un triangle
TC 2	1	4	2	Pas un triangle
TC 3	1	2	4	Pas un triangle
TC 4	5	5	5	Équilatéral
TC 5	?	?	?	Impossible
TC 6	?	?	?	Impossible
TC 7	2	2	3	Isocèles
TC 8	?	?	?	Impossible
TC 9	2	3	2	Isocèles
TC 10	3	2	2	Isocèles
TC 11	3	4	5	Scalène

Figure 12 : Cas de test

- **Échelle :** Pour n conditions, il y a peut-être au plus **2^n variantes** (combinaisons uniques de conditions et actions). Mais, heureusement, il y a d'habitude beaucoup moins de variantes explicites … Les valeurs "Don't care" dans les tables de décision aide à réduire le nombre de variantes. "Don't care" peut correspondre à plusieurs cas : les données sont nécessaires mais n'ont pas d'effet ; les données peuvent être omises ; les cas mutuellement exclusifs (exclusions type-sécurité).

e- Graphes de cause-à-effet : Une technique graphique qui aide à dériver les tables de décision. Vise à supporter l'interaction avec les experts de domaine et l'ingénierie inverse

des spécifications, dans le but de tester. Identifie les causes (conditions de entrées, impulsions) et les effets (sorties, changements dans l'état du système). Les causes doivent être formulées d'une manière pour être soit vraies ou fausses (expression booléenne). Précise explicitement les contraintes (environnementales, externes) des causes et des effets. Aide à sélectionner des sous-ensembles de combinaisons de sous- ensembles "significatifs" d'entrées-sorties et à construire de plus petites tables de décision.

- **Structure des graphes de cause-effect :** Un nœud est tiré pour chaque cause et chaque effet. Les nœuds sont placés sur les côtés opposés de la feuille. Une ligne de la cause à l'effet indique que la cause est une condition nécessaire pour l'effet. Si un simple effet a deux causes ou plus, le rapport logique des causes est annoté par des symboles pour un et logique (^) et ou logique (+). Une cause dont la négation est nécessaire est désignée par un non logique (~). Une cause simple peut être nécessaire pour plusieurs effets ; un effet simple peut avoir plusieurs causes nécessaires. Les nœuds intermédiaires peuvent être utilisés pour simplifier le graphe et sa construction.

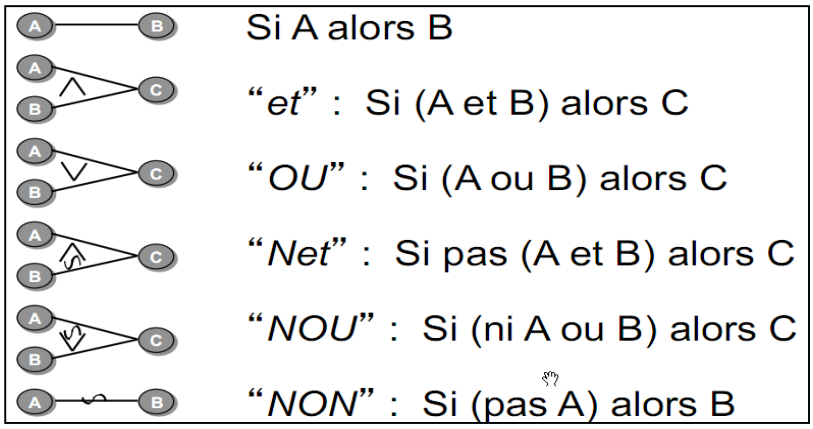


Figure 13 : nœuds graphes de cause-effect

Exemple : prenant le processus de renouvellement d'assurance, son graphe est le suivant :

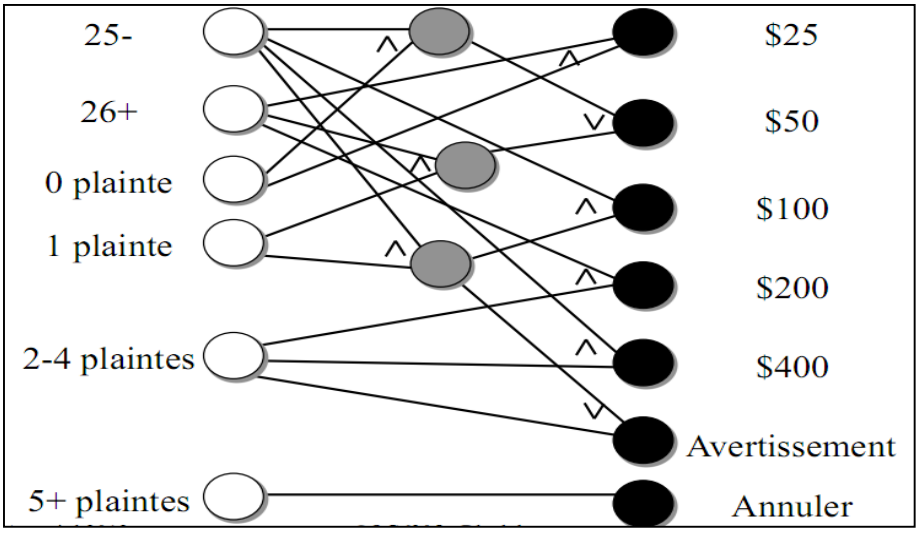


Figure 14 : graphes cause-effect d'assurance

- **Spécification** : d'après le graphe causse-effet, on peut déduire sa table de décision :

	Section condition		Section action		
Variable	**Plaintes**	**Âge**	**Prime augmentée $**	**Envoie d'un avertissement**	**Annulation**
1	0	25-	50	Non	Non
2	0	26+	25	Non	Non
3	1	25-	100	Oui	Non
4	1	26+	50	Non	Non
5	2 à 4	25-	400	Oui	Non
6	2 à 4	26+	200	Oui	Non
7	5+	Tout	0	Non	Oui

Figure 15 : table de décision d'assurance

- **Dériver une table de décision** : Une ligne pour chaque cause ou effet. Les colonnes correspondent aux cas de tests (variables). Nous définissons les colonnes en examinant chaque effet et en listant toutes les combinaisons (conjonctions) des causes qui peuvent mener à cet effet. E.g., deux lignes séparées mènent vers l'effet E3, chacune correspondant à un cas de test, quatre lignes mènent vers E1 mais ne correspondent qu'à deux combinaisons seulement.
- **Discussion** : Le graphe cause-effet peut être utilisé pour générer toutes les combinaisons possibles de causes et vérifier si l'effet correspond à la spécification. Il fournit un test oracle et spécifie les contraintes sur les sorties (effets), aidant à détecter les mauvais états du système et les combinaisons d'action. Si le graphe est trop large, pour chaque combinaison admissible d'effets, on trouve quelques combinaisons de causes qui déclenchent les combinaisons d'effets en remontant vers le graphe. À cause de contraintes additionnelles sur le graphe, on peut être plus restrictif que les tables de décision classiques.

f- Fonctions logique : Un prédicat est une expression qui évalue une valeur Booléenne. Les prédicats peuvent contenir des variables booléennes, des variables non booléennes qui sont comparées avec les opérateurs comparateurs {>, <, =, …}, et les appels de fonction (retournent une valeur booléenne). La structure interne du prédicat est créée par les opérateurs logiques {non, et, ou, …}. Une clause est un prédicat qui ne contient aucun des opérateurs logiques, e.g., (a<b). Les prédicats peuvent être écrits de différentes manières logiques équivalentes (algèbre booléenne). Une fonction logique relie n variables d'entrées booléennes (clauses) à une seule variable de sortie booléenne. Pour rendre les expressions plus simple à lire, nous utiliserons la contiguïté (AB) pour l'opérateur et, + pour (A+B) l'opérateur ou et un ~ pour l'opérateur de négation.

Exemple de la chaudière : mettre en état ou hors état l'ignition d'une chaudière se basant sur quatre variables d'entrée :
NormalPressure(A): la pression dans une limite d'opération sécuritaire ?
CallForHeat (B) : la température ambiante sous le point de consigne ?
DamperShut (C) : le conduit du tuyau d'échappement est fermé ?

ManualMode (D) : sélection du manuel d'opération ?
Fonction logique : **Z = A(B~C+D)** ~> **Table de vérité :**

Numéro du vecteur d' entrées	Normalpressure	CallForHeat	DamperShut	ManualMode	Ignition
	A	B	C	D	Z
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	0

Numéro du vecteur d'enrées	NormalPressure	CallForHeat	DamperShut	ManualMode	Ignition
	A	B	C	D	Z
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	0
15	1	1	1	1	1

Figure 16 : Table de vérité de la chaudière

- **Les avantages**
  - **Simplicité** : ces tests sont simples à réaliser, car on se concentre sur les entrées et les résultats. Le testeur n'a pas besoin d'apprendre à connaître le fonctionnement interne du système ou son code source, qui n'est pas accessible. Cette méthode est donc également non intrusive.
  - **Rapidité :** en raison du peu de connaissances nécessaires sur le système, le temps de préparation des tests est très court. Les scénarios sont relativement rapides à créer et à tester, puisqu'ils suivent les chemins utilisateurs, qui sont relativement peu nombreux selon la taille du système.
  - **Impartialité :** on est ici dans une optique « utilisateur » et non « développeur ». Les résultats du test sont impartiaux : le système marche, ou il ne marche pas. Il n'y a pas de contestation possible, comme par exemple sur l'utilisation de tel processus plutôt qu'un autre selon l'opinion du développeur.

- **Inconvénients**
  - **Superficialité :** étant donné que le code n'est pas étudié, ces tests ne permettent pas de voir, en cas de problème, quelles parties précises du code sont en cause. De plus, les testeurs peuvent passer à côté de problèmes ou vulnérabilités sous-jacentes. Certains problèmes sont également difficilement repérables avec cette méthode, comme par exemple ceux liés à la cryptographie, ou à des aléas (Générateur de nombres aléatoires, Random Number Generator (RNG) de mauvaise qualité. C'est donc l'un des tests les moins exhaustifs.
  - **Redondance** : si d'autres tests sont effectués, il est possible que celui-ci perde grandement de son intérêt, puisque son champ d'action a tendance à être inclus dans celui d'autres tests.

1.3.3 Test Boite Gris : Les tests en « boîte grise » compilent ces deux précédentes approches : ils éprouvent à la fois les fonctionnalités et le fonctionnement d'un système. C'est-à-dire qu'un testeur va par exemple donner une entrée (input) à un système, vérifier que la sortie obtenue est celle attendue, et vérifier par quel processus ce résultat a été obtenu. Dans ce type de tests, le testeur connaît le rôle du système et de ses fonctionnalités, et a également une connaissance, bien que relativement limitée, de ses mécanismes internes (en particulier la structure des données internes et les algorithmes utilisés). Attention cependant, il n'a pas accès au code source !

Ces tests peuvent difficilement être effectués pendant la phase de développement du projet, car elle implique des tests sur les fonctionnalités du programme : celui-ci doit déjà être dans un état proche de final pour que ces tests puissent être pertinents. En effet, pendant les tests en « boîte grise », ce sont surtout des techniques de « boîte noire » qui sont utilisées, puisque le code n'est pas accessible. Cependant, les scénarios sont orientés pour jouer sur les processus sous-jacents, et ainsi les tester également. Bien entendu, la méthode « boîte grise » combine surtout les avantages des méthodes « boîte blanche » et « boîte noire ». On peut cependant noter deux gros **bénéfices** de cette technique :

- **Impartialité** : les tests en « boîte grise » gardent une démarcation entre les développeurs et le testeur, puisque ce dernier n'étudie par le code source et peut s'appuyer sur les résultats obtenus en testant l'interface utilisateur.
- **Intelligence** : en connaissant la structure interne du programme, un testeur peut créer des scénarios plus variés et intelligents, afin d'être certain de tester toutes les fonctionnalités mais également tous les processus correspondants du programme.

En parallèle, l'un des **inconvénients** les plus importants de ces tests est le suivant : le **non exhaustivité** : étant donné que le code source n'est pas accessible, il est impossible avec des tests en « boîte grise » d'espérer avoir une couverture complète du programme.

- **Etude comparative entre test fonctionnelle et test structurelle :**

Test en boîte noire	Test en boîte blanche
C’est une manière de tester le logiciel dans laquelle la structure interne du programme ou du code est caché et rien n’est connu à ce sujet.	C’est une façon de tester le logiciel dans lequel le testeur a une connaissance de structure interne du code ou du programme du logiciel.
Cela se fait principalement par des testeurs de logiciels.	C’est principalement fait par les développeurs de logiciels.
Aucune connaissance de l’implémentation n’est nécessaire.	La connaissance de l’implémentation est requise.
Il peut être appelé test externe du logiciel.	C’est le test interne du logiciel.
C’est un test fonctionnel du logiciel.	C’est un test structurel du logiciel.
Ce test peut être lancé sur la base du document de spécification des exigences.	Ce type de test est entamé après le document de conception détaillée.
Aucune connaissance en programmation n’est requise.	Il est obligatoire de connaître la programmation.
C’est le test de comportement du logiciel.	Ce sont les tests logiques du logiciel.
Cela prend moins de temps.	Cela prend beaucoup de temps.

Figure 16 : étude comparative entre les types de tests

Université de Mohamed Khider – Biskra –
Département d'informatique
Niveau : 1ère Master (GLSD) **Module : TTL**
Année : 2023

TD N° 2

Partie : méthode de test boite blanche

Soit les problèmes très reconnus suivants :

Exercice 1 : vérifier qu'un nombre N est pair ou impaire.

Exercice 2 : calculer la puissance d'un nombre N.

Exercice 3 : équation 2em degré

Travail demandé : Appliquer la méthode de test Structurels (Boîte blanche) pour résoudre ces problèmes suivants comme suite :

- Ecrire votre solution sous forme algorithme ou pseudo code
- Elaborer le graphe de flux de contrôle
- Calculer la complexité cyclomatique V(G)
- Identifier les chenins possibles
- Proposer des jeux de test a votre choix
- Vérifier chaque chemin de graphe
- Raffiner votre solution

Partie 2 : méthode de test boite noire

Spécification 1 : la température d'eau peut varie de -40 à +100 Celsius.

- En dessous de 0 degré Celsius, le system met en route le chauffage antigel.
- Au-dessus de 30 degré Celsius, le système met en route le refroidissement.

Spécification 2 : dans une université, le système autorise les étudiants à utiliser de l'espace disque en fonction de leurs projets.

- S'ils ont utilisés tout leur espace alloué, ils sont autorisés en accès restreint (ils peuvent lire et supprimer des fichiers mais pas en créer).
- ils doivent toutefois être connectés avec leur identifiant et leur mot de passe

Spécification 3 : le distributeur de billets automatique, parmi plusieurs solutions choisir une et appliquer le travail demandé.

Spécification 4 : Il existe d'autre méthodes de test dans la technique boite noire tel que :

- Tests de transition d'états

- Tests de cas d'utilisation

Travail demandé : Etudier l'essentiel de ces méthodes, avec des exemples**.**

-Partie N°2-

1. **Autres Méthodes de Test**
   - **Test de robustesse :** C'est une technique de test qui a pour objectif de déterminer si le système se comporte comme attendu en présence de conditions d'exécution erronées ou non prévues. Exemple **:** Application industrielle (incendie, courant, fumée. etc).
   - **Test de performance :** C'est une technique de test qui a pour but de déterminer les performances d'un système dans une situation particulière (temps de réponse d'un logiciel, utilisation des ressources, etc.). Exemple **:** Application web/mobile. Jeux
   - **Test de sécurité :** Cette technique de test vise à rechercher par des tests les vulnérabilités (points faibles) d'un système. Exemple Application bancaire, DBA, etc.
2. **Rapport de test :** Pour chaque phase de test l'équipe doit élaborer un rapport de tests. Ce rapport est la synthèse des actions menées suivantes : Exécution des fiches de tests (effectuer les actions décrites). Analyser les résultats obtenus : comparer les résultats attendus avec les résultats obtenus. Les éléments de mesure sont très importants ! Emettre des fiches de non-conformité si nécessaire (ces fiches sont aussi appelées fiches d'anomalie, fiches de bug). Il s'agit de coupler intelligemment la gestion des tests et la gestion des corrections (incidents). Consigner tous les résultats d'exécution de tests. Rédiger des comptes rendus de tests. La somme de ces comptes rendus constituera le rapport de tests.

Plan de test (Nom du produit)	**6.1 Cadre principal**
Préparé par (Noms de ceux qui ont préparé) (Date)	**6.2 Poste de travail**
TABLE DES MATIÈRES (TOC)	**7. Calendrier des tests**
1. INTRODUCTION	**8. Procédures de contrôle**
2. OBJECTIFS ET TÂCHES	**9. Fonctionnalités à tester**
2.1 Objectifs	**10. Fonctionnalités à ne pas tester**
2.2 Tâches	**11. Ressources/Rôles et responsabilités**
3. PORTÉE	**12. Horaires**
4. Stratégie de test	**13. Départements significativement impactés (SID)**
5. Configuration matérielle requise	**14. Dépendances**
6. Exigences environnementales	**15. Risques/hypothèses**
	16. Outils
	17. Approbations

Figure 1 : exemple du contenu d'un rapport de test

3. **Les niveaux de test :** A chaque **phase** (niveau) de développement logiciel est associée son propre type de test comme suit :

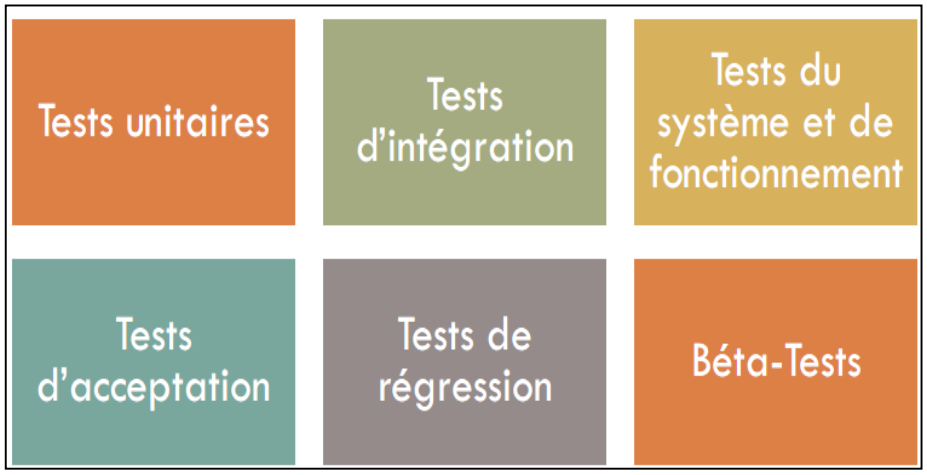
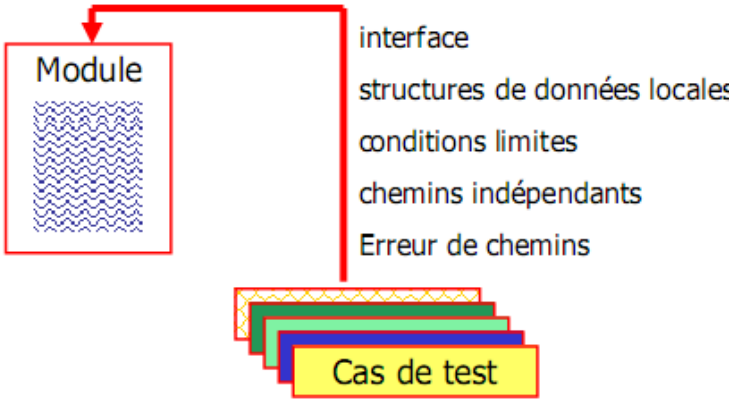
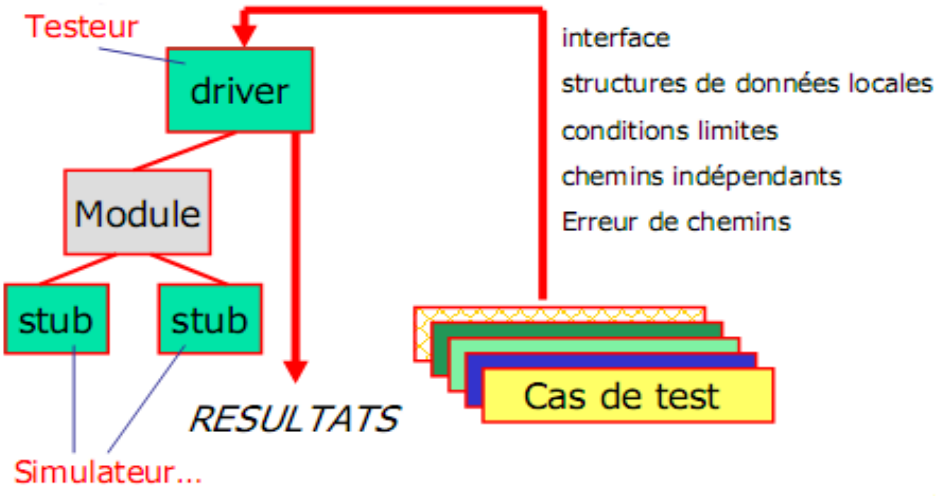


Figure 2 : les six niveaux de test

3.1 test unitaire



➢ **Environnement du test unitaire**

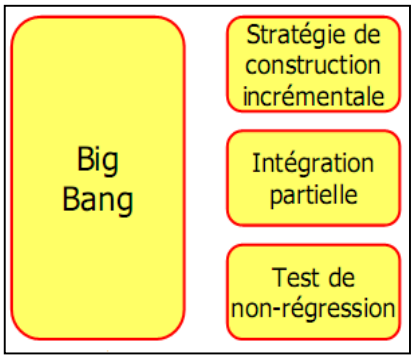


3.2 Test d'intégration

Si tous les modules marchent bien séparément, pourquoi douter qu'ils ne marcheraient pas ensemble ?

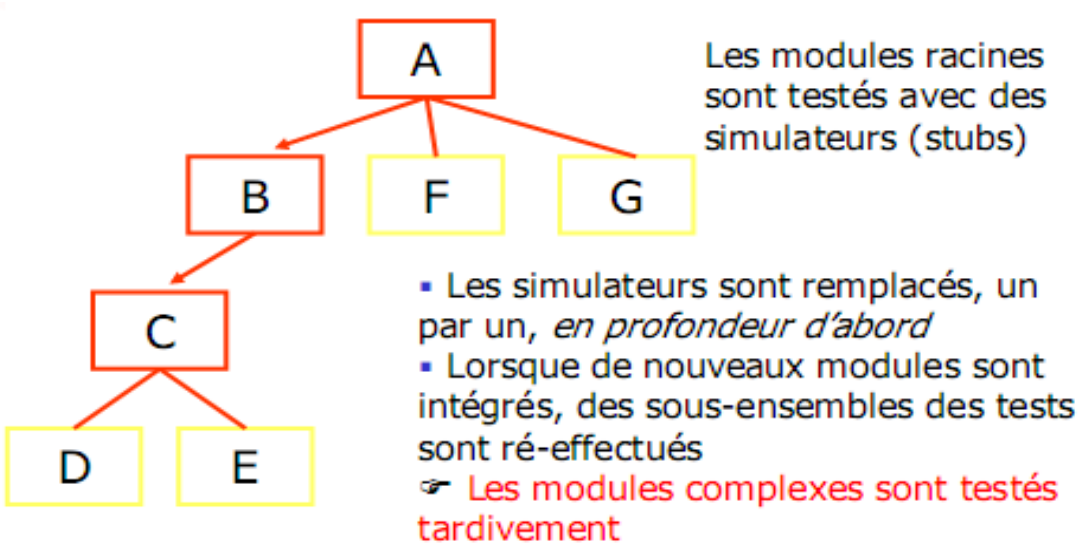
Réunir les modules : **Interfacer**

- Big bang (interface)
- Stratégie de construction incrémentale
- Intégration partielle

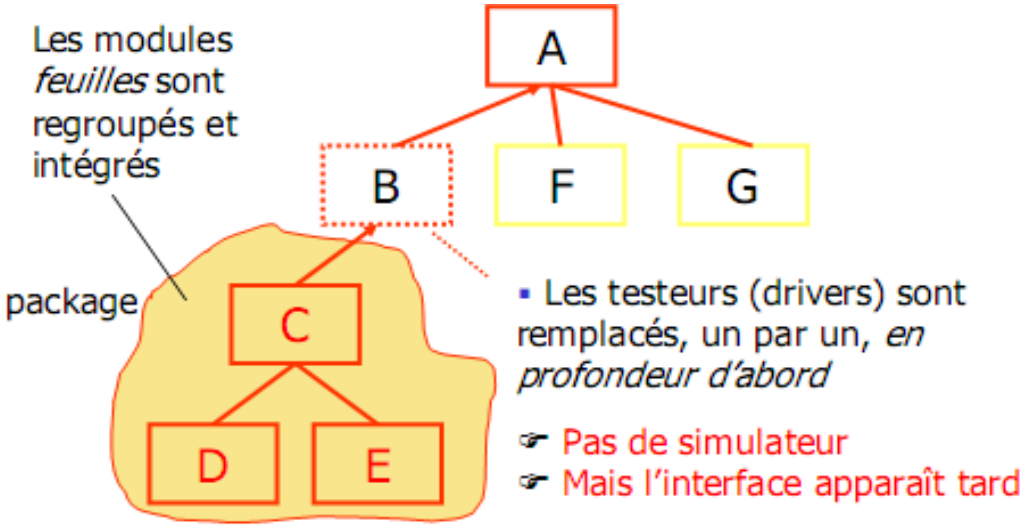


- Test de non régression : permet de vérifier que les changements effectués durant la mise au point ne permettent pas d'introduire d'autres défauts !

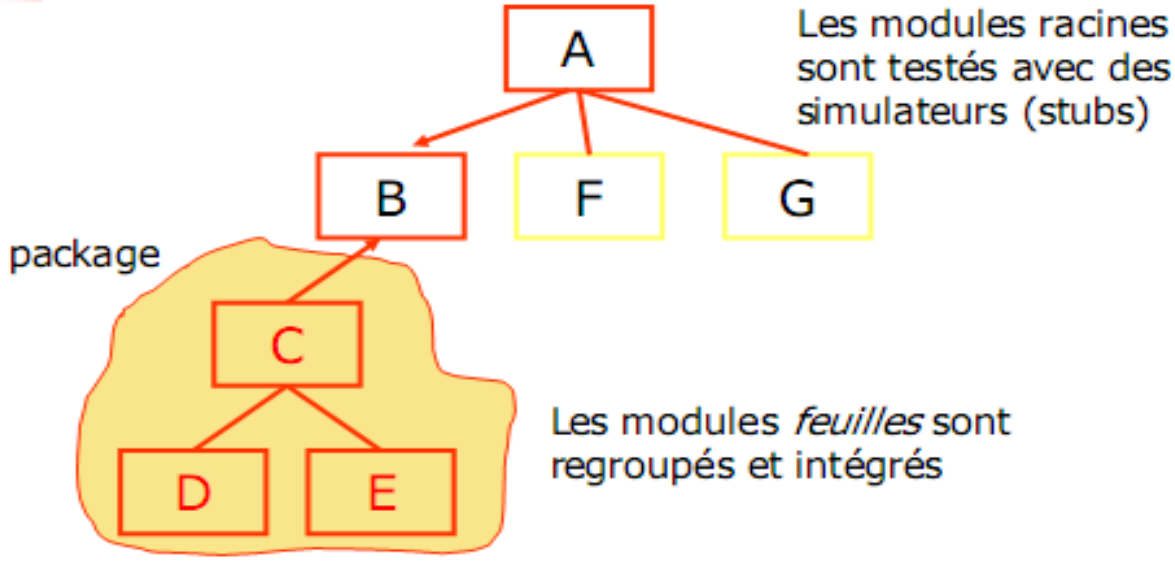
➢ **Intégration descendante**



➢ **Intégration ascendante**



➢ **Intégration en Sandwich (mixte)**



3.3 Test Système : Le test système a pour but de vérifier que le logiciel est conforme à ses spécifications. En d'autres termes cela revient à s'assurer par des tests que le logiciel remplit bien les fonctionnalités attendues. Cette phase vient après celle de l'intégration du logiciel. La méthode de test boite noire est généralement utilisée pour ce type de test.

3.4 Test d'Acceptation Cette phase de test est l'une des plus importantes, car elle fait intervenir les exigences du client. En effet, elle a pour but de vérifier si le logiciel est conforme, en plus de ses spécifications, aux différentes exigences fixées par le client.

3.5 Test Alpha : Le test alpha est réalisé après les tests d'acceptation. Il a pour objectif de faire tester le logiciel en interne, par des utilisateurs qui n'ont pas participé au développement du projet.

3.6 Test Bêta : Le test bêta est réalisé après le test alpha. Il a pour but de faire tester le logiciel par un échantillon de personnes, le plus souvent externes à l'entreprise (des potentiels utilisateurs par exemple). Le suivant figure synthétise les niveaux de test :

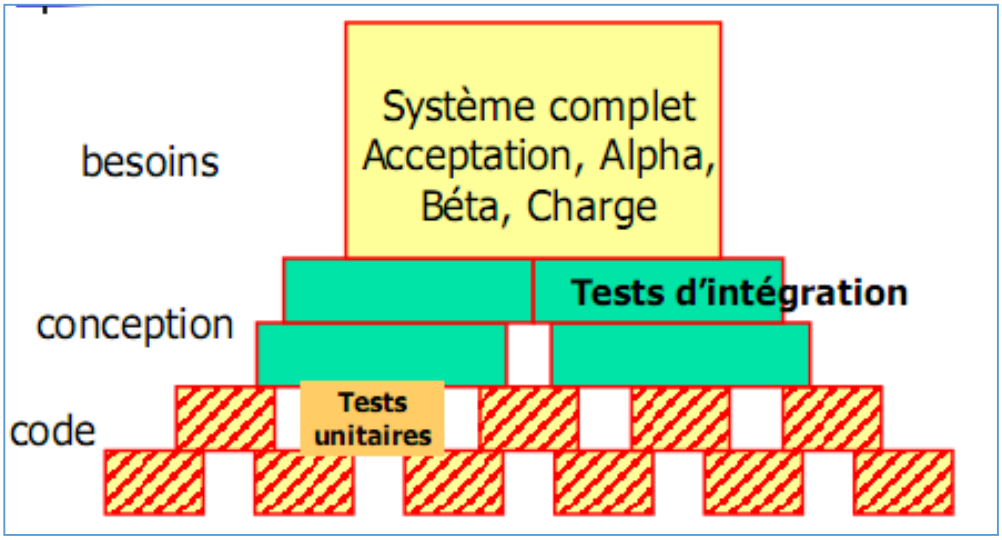


Figure 3 : Synthèse des niveaux de tests

➢ **Résumé :** les types de tests se positionnent dans un cycle en V, comme montre la figure :

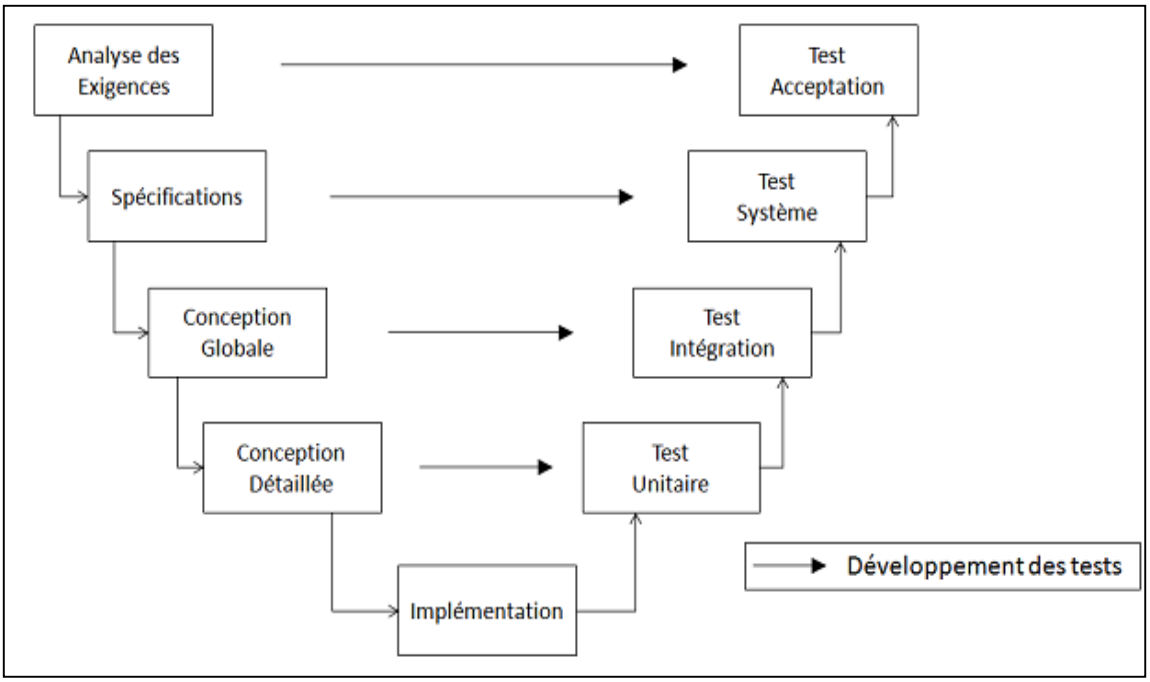


Figure 4 : Positionnement des types de tests dans un cycle en V

➢ **Acteurs de test** : Les principaux acteurs et activités qui interviennent tout au long du test :

Niveaux de Test	Méthode de Test	Source utilisée	Qui teste ?	But
Unitaire	Boite Blanche + Boite Noire	Conception détaillée ou Code source	Programmeurs	Tester des portions élémentaires dans le modèle utilisé (ex, une classe)
Intégration	Boite Blanche + Boite Noire	Conception Architecture	Programmeurs ou Testeurs	Tester des combinaisons de plusieurs portions
Système	Boite Noire	Spécifications	Testeurs	Tester le système par rapport à ses spécifications
Acceptation	Boite Noire	Exigences (clients)	Du coté du client	Tester le produit par rapport aux exigences du client
Alpha	Boite Noire	Produit	Testeurs n'ayant pas participé au projet	Tester le produit par des testeurs indépendants
Beta	Boite Noire	Produit	Utilisateurs potentiels	Tester le produit par des utilisateurs potentiels

Figure 4 : acteurs de test

➢ **Degré de tests :** il se traduire selon six niveaux cimentaires :

- **Fatal :** impossible de contenir les tests à causés de la sévérité des défaillances
- **Critique :** les tests peuvent contenir mais l'application ne peut passer en mode production

- **Majeur :** l'application peut passer en mode production mais des exigences fonctionnelles importantes ne sont pas satisfaites
- **Medium :** l'application peut passer en mode production mais des exigences fonctionnelles sans très grand import ne sont pas satisfaites.
- **Mineur :** l'application peut passer en mode production, la défaillance doit être corrigée mais elle sans import sur les exigences métier
- **Cosmétique** : défaillance mineurs relatives à l'interface (couleurs, police,..) mais n'ayant pas une relation avec les exigences du client.

4. **Les activistes liées au projet de tests :** L'activité de tests est un projet à part entière.
   C'est la raison pour laquelle nous retrouvons l'ensemble des caractéristiques d'un projet

   - Planifier les tests : c'est-à-dire prévoir où et en quelle quantité porter les efforts en personne et en temps. Métriques : estimation des charges, définition des métriques, définition des environnements matériels et logiciels, définition de la campagne, du plan et des livrables.

   - Spécifier les tests : c'est-à-dire préciser ce que l'on attend des tests en termes de détection d'erreurs (fonctionnel ou non fonctionnel par exemple). Métriques : Les techniques et les outils à utiliser ou à ne pas utiliser, les parties du logiciel sur lesquelles porteront les tests ;

   - Concevoir les tests : c'est-à-dire définir les scénarios de tests, appelés aussi cas de test, permettant de mettre en évidence des défauts recherchés par les tests, en fonction de leurs spécifications ; il s'agit aussi de préciser les environnements d'exécution de ces tests. Métriques : organisation du référentiel, identification des conditions de tests, traçabilité, cas de tests, données de tests, procédures de tests, scénarios.

   - Établir les conditions de tests : c'est-à-dire prévoir pour chaque scénario de tests les **jeux** de valeurs à fournir au logiciel pour réaliser ce scénario. Métriques suivi, exécution et reporting.

   - Définir les conditions d'arrêt d'une campagne de tests : c'est-à-dire définir, en relation avec la phase de planification, ce qui permettra de décider l'arrêt ou la poursuite des tests. Métriques : Evaluation des risques et Gestion des incidents.

   - Contrôler les résultats : c'est-à-dire comparer les données fournies par l'exécution des tests par rapport aux données attendues ou constatées lors de phases précédentes de tests.

   - Tracer les tests vis-à-vis des exigences grâce à une matrice de traçabilité : c'est-à-dire analyser en quoi les tests fournit une exhaustivité de la recherche d'erreurs dans les attendus du logiciel. Cette matrice permet de vérifier qu'à toute exigence correspond au moins un scénario de test (mesure de la complétude des tests) et, à l'opposé, elle permet également de vérifier qu'un scénario de test sert à valiser au moins une exigence (mesure de l'efficacité des tests). Métriques (recette ou arrêt des tests) et Amélioration des processus.

5. **Approches des jeux de test : Tests exhaustifs** : L'objet principal est d'analyser le comportement d'un logiciel dans un environnement donné. Ainsi, il ne sert a priori à rien de tester le bon fonctionnement de la gestion du système de freinage d'une automobile pour une vitesse de 1000 km/h. En général, tester un programme de façon exhaustive est impossible. Il faut choisir un sous-ensemble des tests qui maximise la probabilité de détecter les erreurs. Il est possible d'utiliser des tests aléatoires, mais leur efficacité est faible pour tester le comportement attendu. Une meilleure approche : déterminer un ensemble de tests fonctionnels qui seront complémentés de tests structurels.

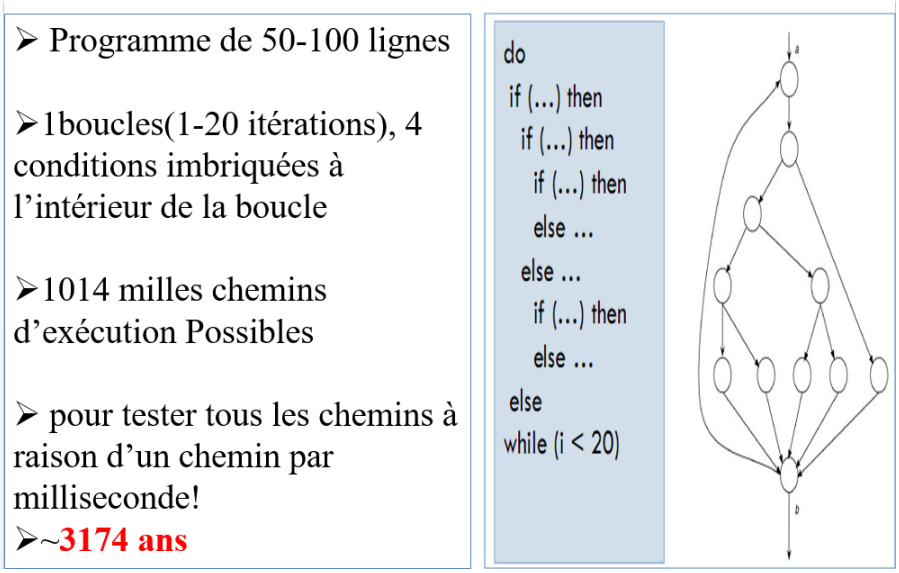


Figure 4 : test exhaustive

- **Spécification** : Un programme prend en entrée trois entiers. Ces trois entiers sont interprétés comme représentant les longueurs des côtés d'un triangle. Le programme rend un résultat précisant s'il s'agit d'un triangle scalène, isocèle ou équilatéral. Produire une suite de cas de tests pour ce programme, 13 cas de test :

1. Cas scalène valide (1,2, 3 et 2,5,10 ne sont pas valides)

2. Cas équilatéral valide

3. Cas isocèle valide (2,2, 4 n'est pas valide)

4. Cas isocèle valide avec les trois permutations (e.g. 3,3, 4; 3,4,3; 4,3,3)

5. Cas avec une valeur à 0

6. Cas avec une valeur négative

7. Cas ou la somme de deux entrées est égale à la troisième entrée

8. cas pour le test 7 avec les trois permutations

9. Cas ou la somme de deux entrées est inférieur à la troisième entrée

10. 3 cas pour le test 9 avec les trois permutations

11. Cas avec les trois entrées à 0

12. Cas avec une entrée non entière

13. Cas avec un nombre erroné de valeur (e.g. 2 entrées, ou 4)

Chacun de ces 13 tests correspond à un défaut constaté dans des implantations de cet exemple triangle. La moyenne des résultats obtenus par un ensemble de développeurs expérimentés est de 7.8 sur 13. La conception de tests est une activité complexe, a fortiori sur de grandes applications

6. **Les 7 principes généraux des tests** Un nombre de principes de test ont été suggérés au cours des 40 dernières années et offrent des indications communes à tous les tests.

Principe 1 : Les tests montrent la présence de défauts :Les tests peuvent prouver la présence de défauts, mais ne peuvent pas en prouver l'absence. Les tests réduisent la probabilité que des défauts restent cachés dans le logiciel mais, même si aucun défaut n'est découvert, ce n'est pas une preuve d'exactitude.

Principe 2 : Les tests exhaustifs sont impossibles : Tout tester ! (toutes les combinaisons d'entrées et de préconditions) n'est pas faisable, sauf pour des cas triviaux. Plutôt que des tests exhaustifs, nous utilisons l'analyse des risques et des priorités pour focaliser les efforts de tests.

Principe 3 : Tester tôt : Pour trouver des défauts tôt, les activités de tests devraient commencer aussi tôt que possible dans le cycle de développement du logiciel ou du système, et devraient être focalisées vers des objectifs définis.

Principe 4 : Regroupement des défauts :L'effort de test devrait être fixé proportionnellement à la densité des défauts prévus et constatés dans les différents modules. Un petit nombre de modules contiennent généralement la majorité des défauts détectés lors des tests pré-livraison, ou affichent le plus de défaillances en opération.

Principe 5 : Paradoxe du pesticide : Si les mêmes tests sont répétés de nombreuses fois, il arrivera que le même ensemble de cas de tests ne trouve plus de nouveaux défauts. Pour prévenir ce "paradoxe du pesticide", les cas de tests doivent être régulièrement revus et révisés, et de nouveaux tests différents doivent être écrits pour couvrir d'autres chemins dans le logiciel ou le système de façon à permettre la découverte de nouveaux défauts.

Principe 6 : Les tests dépendent du contexte : Les tests sont effectués différemment dans des contextes différents. Par exemple, les logiciels de sécurité critique seront testés différemment d'un site de commerce électronique.

Principe 7 : L'illusion de l'absence d'erreurs : Trouver et corriger des défauts n'aide pas si le système conçu est inutilisable et ne comble pas les besoins et les attentes des utilisateurs.

Université de Mohamed Khider – Biskra –

Département d'informatique

Niveau : 1ère Master (GLSD) **Module : TTL**

Année : 2023

TD N° 3 :

Rapport de test : Choisir le meilleur exemple afin d'éditer son rapport de test le plus adéquat, qui respecte le plan suivant.

(Nom du produit)

Préparé par :

(Noms de ceux qui ont préparé)

(Date)

TABLE DES MATIÈRES (TOC)

1. INTRODUCTION

2. OBJECTIFS ET TÂCHES

2.1 Objectifs

2.2 Tâches

3.0 PORTÉE

4. Stratégie de test

4.1 Test alpha (test unitaire)

4.2 Tests du système et de l'intégration

4.3 Performances et tests de résistance

4.4 Test d'acceptation par l'utilisateur

4.5 Tests par lots

4.6 Tests de régression automatisés

4.7 Test bêta

5. Configuration matérielle requise

6. Exigences environnementales

6.1 Cadre principal

6.2 Poste de travail

7. Calendrier des tests

8. Procédures de contrôle

9. Fonctionnalités à tester

10. Fonctionnalités à ne pas tester

11. Ressources/Rôles et responsabilités

12. Horaires

13. Départements significativement impactés (SID)

14. Dépendances

15. Risques/hypothèses

16. Outils

17. Approbations

Chapitre 3 :

Outils de Test

1. Classification des outils de test :

Les outils de teste sont classés en plusieurs groups selon le type de système à tester :

1.1 Le premier groupe : concerne les outils d'aide à la réalisation des tests.

- Capture et réexécution des scripts réalisés via une IHM.
- Sauvegarde des tests et des résultats associés.
- Génération de scripts de tests en fonction des langages et des plateformes

- **Outils**
  - Mercury WinRunner et Quicktest Pro de Mercury Quality Center
  - QARUN de MicroFocus
  - ABBOT (open source)
  - Rational robot de IBM
  - Irise studio de Irise

1.2 Le second groupe : concerne les outils de campagne de tests.

Les outils de gestion des plans et campagnes de test servent à définir, organiser et conduire les campagnes de tests. Ils doivent donc s'interfacer avec tous les outils qui interviennent dans les tests.

- **Outils :**
  - Testdirector de Mercury Quality Center
  - Salomé TMF (open source)
  - Test Manager de SoftEdition.Net
  - QaDirector de MicroFocus

1.3 Le troisième groupe : concerne les tests fonctionnels (boite noire)

Les tests concernent l'analyse des spécifications du logiciel, sans tenir compte du code source. Le périmètre des contrôles englobe : les interfaces utilisateurs, les API, la gestion des bases de données, la sécurité et le réseau. Il vérifie la conformité du fonctionnement d'un système vis-à-vis des exigences de l'utilisateur (se réfère au cahier des charges).

- **Outils**
  - LEIRIOS Test Generator de LEIROS
  - Conformiq Test Generator de Conformiq Software
  - Mercury Functional Testing et Mercury Service Test
  - Test Vector Generation de T-VEC
  - AutoTester ONE de AutoTester
  - QACenter Enterprise Edition de Compuware
  - Quality Forge de TestSmith
  - Selenium (open source)
  - Rational Robot d'IBM
  - iRise Application Simulator de iRise
  - Mercury Business Process Testing de Mercury Interactive
  - TestView, WebFT de Radview
  - Seapine SQA de Seapine
  - SilkTest de Segue

• Visual WebTester de Softbees -
• eValid de Software Research

1.4 Le quatrième groupe : concerne les tests structurels (boite blanche)

Ces outils permettent de valider ce que fait le logiciel à tester. Ils sont donc complémentaires aux outils de tests fonctionnels qui vérifient eux, ce que doit faire le logiciel. Ces pourquoi des éditeurs ont créé des suites comprenant ces deux types de tests.

- **Outils :**
  • C++TEST, .TEST, JTEST, SOATEST et INSURE++ de PARASOFT
  • RATIONAL TEST REALTIME de IBM
  • XUNIT : JUNIT, PHPUNIT, CPPUNIT, PYUNIT, ETC

1.5 Le cinquième groupe : concerne la performance des logiciels développés, quel que soit l'environnement.

• Le test de montée en charge.
• La simulation d'un environnement spécifique
• L'évolution agressive de l'accès aux ressources

- **Outils:**
  • Wapt de Softlogica (test de charge)
  • LoadRunner de Mercury quality center – (test de stress et de charge)
  • Siege (open source) (test de charge)
  • jMeter (open source) du groupe apache (test de performance)
  • QaLoad de MicroFocus (test de charge)
  • Performance Center de EMBARCADERO (test de performance)
  • Web Performance Load Tester de WebPerformance, inc (test de charge)

2. Application JUnit : un Framework

2.1 Principe : JUnit est un framework open source pour le développement et l'exécution de tests unitaires, manuelles et automatisables, pour Java. Nous avons 2 types pour effectuer les tests unitaires JUnit.

- **Test Manuel :** c'est de lancer le test manuellement sans aucun soutien de l'outil, dans ce cas le test ne sera pas fiable et beaucoup de temps sera perdu.
- **Test automatisé :** C'est l'exécution des cas de test par le soutien de l'outil, dans ce cas le test est rapide et plus fiable. JUnit fourni les propretés suivantes :
- Enchaîne **l'exécution** des méthodes de test définies par le testeur
- Facilite la **définition** des tests grâce à des **assertions** (tester les résultats attendus), des méthodes d'initialisation et de finalisation.
- Permet en un seul **clic** de savoir quels tests ont échoué ou réussi
- JUnit n'écrit pas les tests !
- Il ne fait que les lancers.
- JUnit ne propose pas de principes/méthodes pour structurer les tests

Unit définit deux sortes de versions :
Versions initiales < 4
• utilisation de conventions de nommage
• paramétrage par spécialisation parla classe (JUnit.framework.TestCase)
Versions >= 4
• utilisation d'annotations (@)
• de nouvelles fonctionnalités dans JUnit 4

JUnit verdicts (jugements) sont définis grâce aux assertions placées dans les cas de test :
Pass (vert) : pas de faute détectée
Fail (rouge) : échec, on attendait un résultat, on en a eu un autre
Error : le test n'a pas pu s'exécuter correctement (exception inattendue, ...)

2.2 Etapes de construction des tests sous Junit

Aucune entrée de table des matières n'a été trouvée.Ecrire la classe a testée dans un projet Java sous Eclipse.
1. Accédez à votre projet Java
2. Créer un nouveau dossier source nommé test (dans ce même projet)
3. Créer un nouveau package (il a le même nom que le package de la classe à tester)
4. Choisissez la classe que vous voulez tester,
5. Allez dans Fichier /New ... / JUnitTest Case
6. Sélectionner les méthodes à tester (ou toutes)
7. Construire un objet qui manipule la méthode à tester.

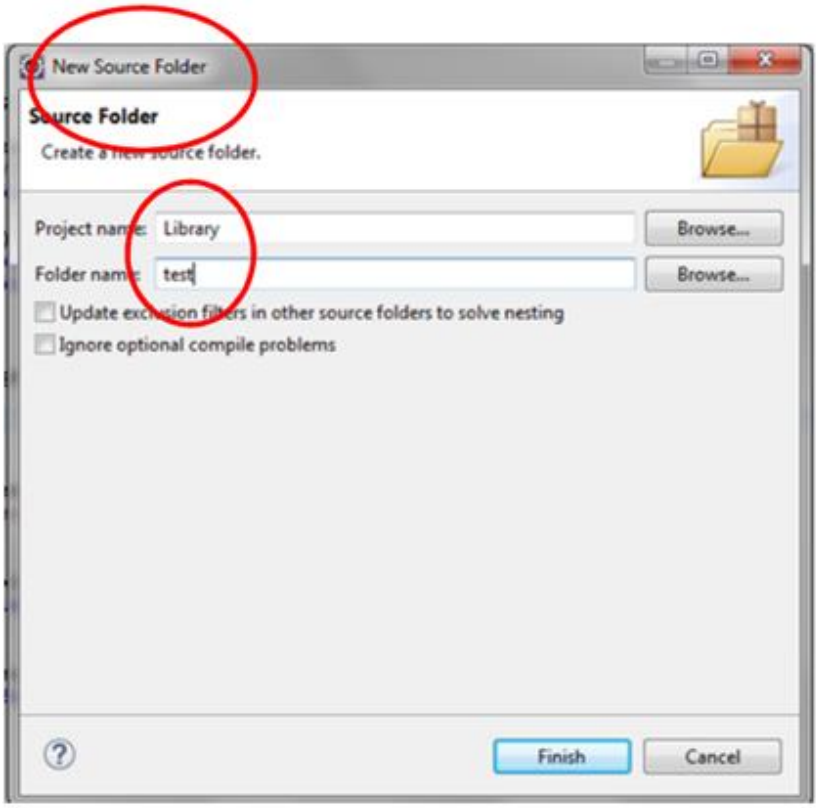
Pour exécuter les tests :
Choisissez Run /Run As /JUnit Test

Démonstration : soit la classe Etudiant en java :

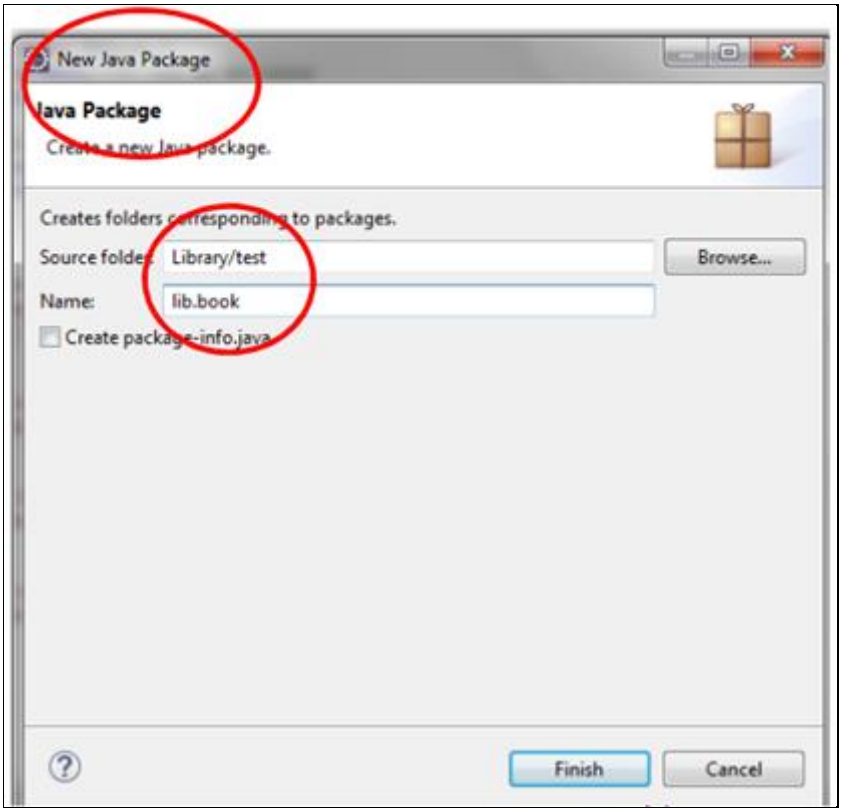
```
package mediateheque;
import java.util.ArrayList;
import java.util.Date;

public class Etudiant {
    String nom;
    String prenom;
    int age;
    float moy, MG;
    Date Dnaissance;
        ArrayList <float> note= new ArrayList <float> ();

    public Etudiant(String nom, String prenom, int age, float moy, Date dnaissance) {
        this.nom = nom;
        this.prenom = prenom;
        this.age = age;
        this.moy = moy;
        Dnaissance = dnaissance;
    }
    public void lectureNote ()
    {}

    float calculerMG (ArrayList note)
    {return  (MG);
    }
```

Figure 1 : Classe Etudiant

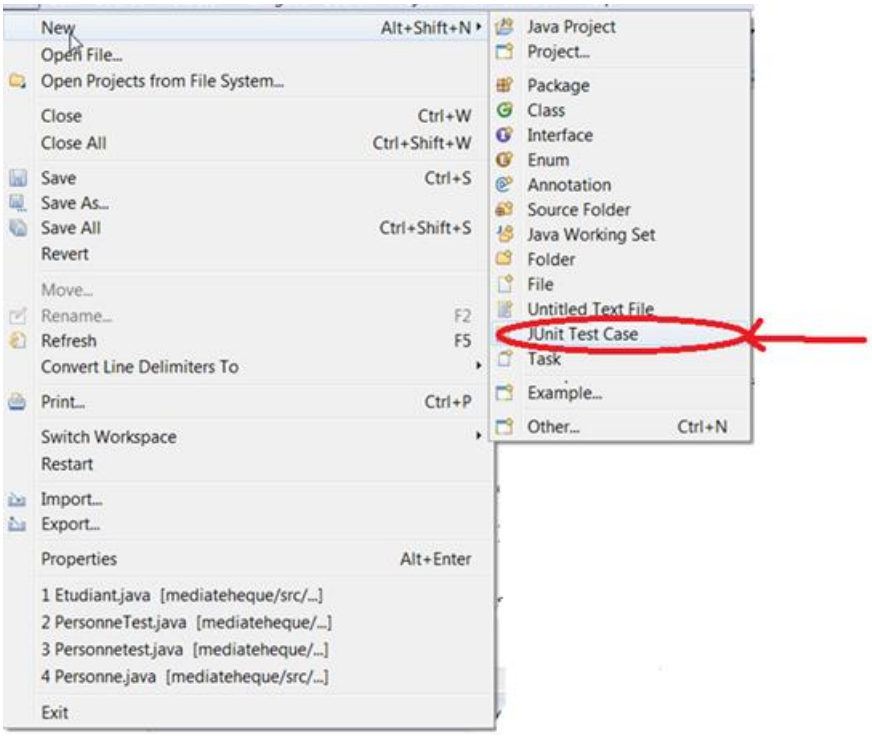
- **Création de nouveau dossier nommé « test »**



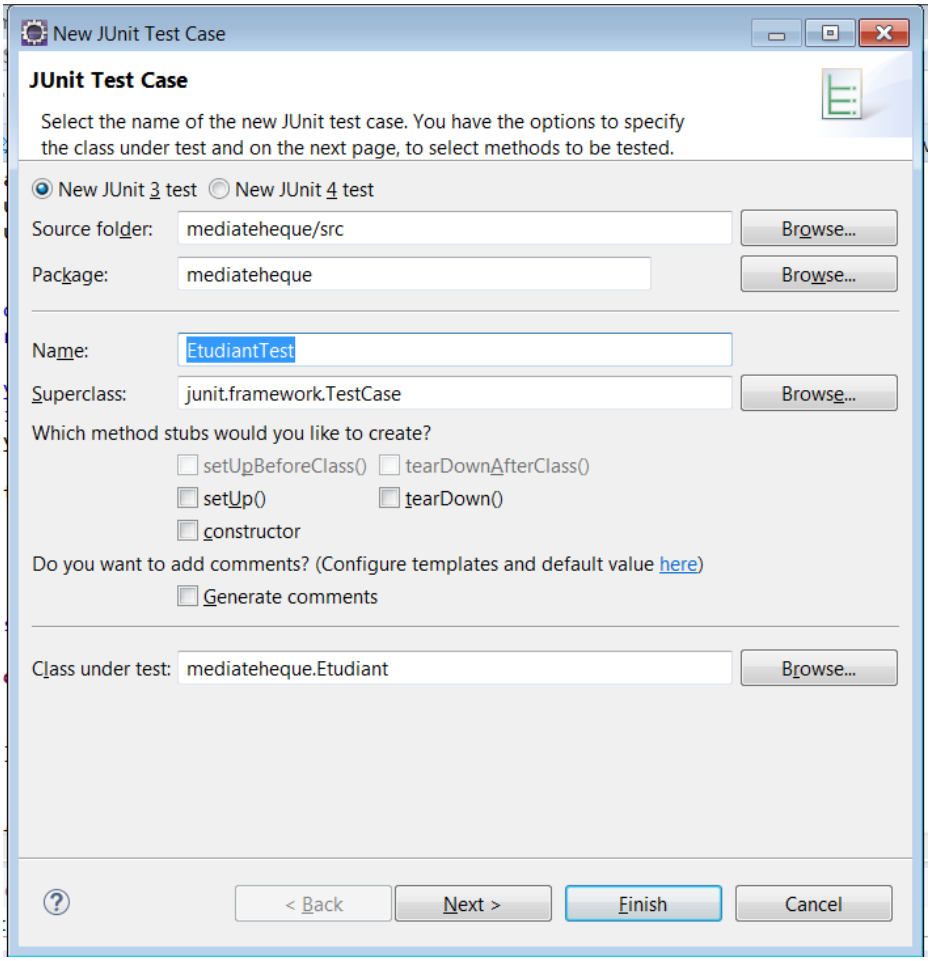
- **Création d'un package porte le même nom que celui à testé**



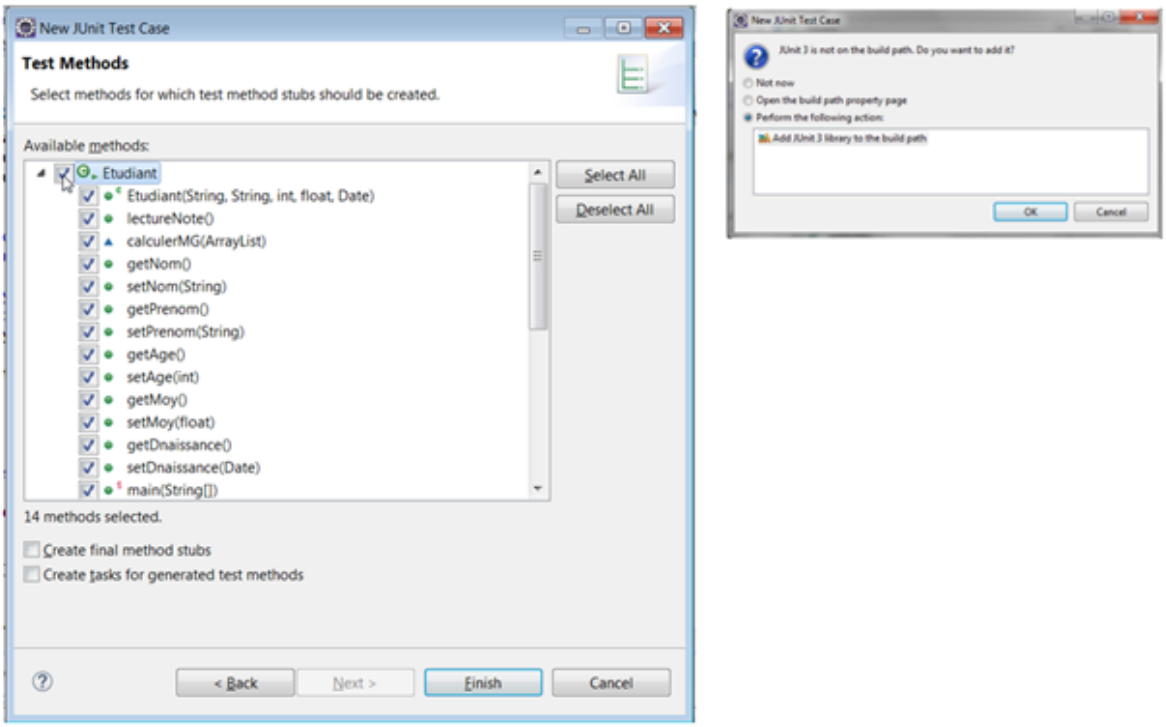
➢ **Création d'une classe de test à partir des classes java à testées**



➢ **La classe de teste porte le même nom que la classe java à testée**



- **Après la création de classe, il faut créer les méthodes de test :**

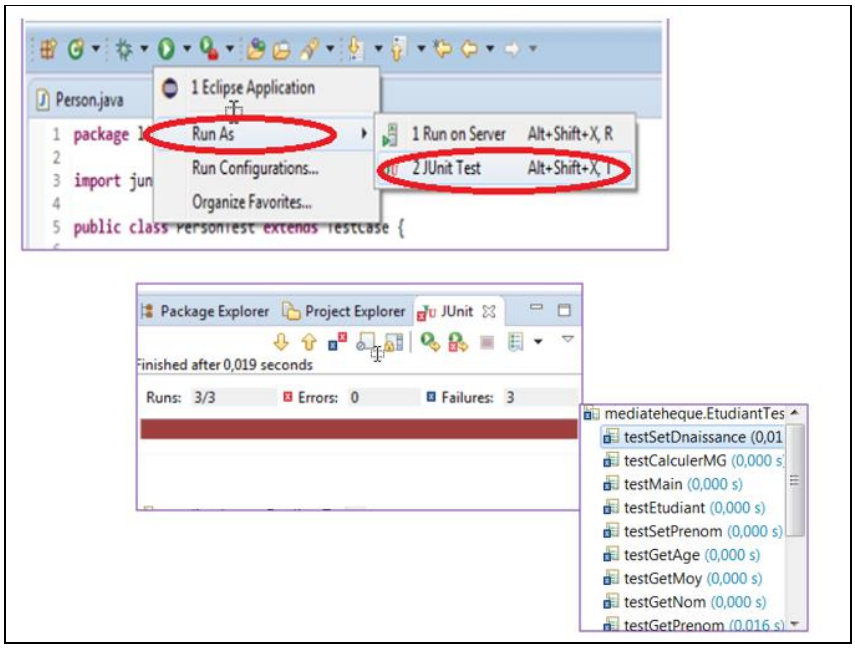


- **JUnit va génère les squelettes des classes de tests :**

```java
package mediatheque;

import junit.framework.TestCase;

public class EtudiantTest extends TestCase {

    public void testEtudiant() {
        fail("Not yet implemented");
    }

    public void testLectureNote() {
        fail("Not yet implemented");
    }

    public void testCalculerMG() {
        fail("Not yet implemented");
    }

    public void testGetNom() {
        fail("Not yet implemented");
    }

    public void testSetNom() {
        fail("Not yet implemented");
    }
```

Remarque : Pour manipuler une méthode de test il faut créer un objet qui manipule la méthode à testée

- **L'exécution de cas de test se fait par JUnit Test**



2.3 Méthodes d'assertion

Méthode	Rôle et exemple
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. Exemple utilisant une méthode : assertEquals ("mohamed dib ", p.getName()); Exemple utilisant un attribut : assertEquals ("l'incendie", b.tilel);
assertTrue ()	Vérifier que la valeur fournie en paramètre est vraie. Exemple utilisant une méthode : assertTrue (dic.isEmpty()); Exemples utilisant un attribut : assertTrue(dic.keys instanceof ArrayList); assertTrue (b.author instance of String); Exemple utilisant une expression : assertTrue (p.getMaximumBooks()==3);
assertFalse ()	Vérifier que la valeur fournie en paramètre est fausse. Exemple utilisant une méthode : assertFalse (dic.isEmpty()); Exemple utilisant une expression : assertFalse (p.getMaximumBooks() >3);

assertNull ()	Vérifier que l'objet fourni en paramètre soit null. Exemple : assertNull (b2.getPerson()) ;
assertNotNull ()	Vérifier que l'objet fourni en paramètre ne soit pas null. Exemple : b1.setPerson (p1) ; assertNotNull (b1.getPerson()) ;
assertSame ()	Vérifier que les deux objets fournis en paramètre font référence à la même entité. Exemples identiques : Exemple : assertSame ("Les deux objets sont identiques", obj1, obj2); //Obj obj1=new Obj (); Obj obj2=obj1 ; Exemple : assertTrue ("Les deux objets sont identiques ", obj1 == obj2);

2.4 Application :<u>Exercice corrigé : les tests sous JUnit</u>

Soit la **classe Pile** qui est caractérisée comme montre la figure en face. Afin d'appliquer les tests sous **JUnit** : réaliser les suivantes méthodes de test par les assertions les plus pertinentes :

Pile
Int taille = 10 ;
PileVide ( ) : Boolean ; HauteurPile ( ) : Int ; PilePleine ( ) : Boolean ; Empiler (Int : a) : Int; Dépiler ( ) : Int; Succ (Int: s): Int ;

2 **Vérifier** le **type** de l'attribut «taille » selon son constructeur.
3 **Dépiler** dans une pile **vide** ! (il faut avoir une plie vide), aussi :
Vérifier l'auteur et le **successeur** de la pile dans cette situation.
4 **Empiler** dans une pile **pleine** ! (il faut avoir une plie pleine), aussi :
Vérifier l'auteur et le **successeur** de la pile dans la nouvelle situation.

Il existe une variété des solutions possibles

- **Solution de la question 1 :**

```
import java.util.Pile;
import junit.framework.TestCase;
public class PileTest extends TestCase {
public void testPile(){
        Pile p = new Pile ();
    assertTrue (p.taille instanceof Int);
    assertTrue (p.gettaille()==10);    }
```

- **Solution de la question 2 :**

```
 public void testDépiler (){
     Pile p1 = new Pile ();
     Int a= p1.Dépiler() ;
```

assertEquals(**0**,a);
assertEquals(**true**, p1.PileVide());
assertEquals(**false**, p1.PilePleine()) ;
assertEquals(0, p1.HauteurPile());
*assertTrue(*p1.PileVide()*);*
*assertFalse(*p1.PilePleine()) ;

Int b=p1.HauteurPile();
assertTrue(b==0);
*assertTrue(*p1.*position()==0);*
Int c=Succ(b) ;
Int c=Succ(p1. HauteurPile();) ;
*assertTrue(*p1.Succ*(0)==1);*
*assertTrue(*p1.Succ*(*p1.*position)==1);*
assertEquals(c, p1.position+1);}

- **Solution de la question 3 :**

public void testEmpiler (){

Pile p2 = **new** Pile ();

Int a= p2. Empiler (10) ; // pile pleine !

assertEquals(a,11);

assertEquals(**false**, p2.PileVide());

assertEquals(**true**, p2.PilePleine()) ;

assertEquals(10, p2.position);

assertEquals(10, p2. Hauteur());

*assertFalse(*p2.PileVide()*);*

*assertTrue(*p2.PilePleine()) ;

Int b=p2. HauteurPile();

*assertTrue(*p2.*position()==10);*

*assertFalse(*b<10);

Int c=Succ(p2.*position()*) ;

Int c=Succ(p2.HauteurPile()) ;

*assertTrue(*p2.position==*10);*

*assertTrue(*c=11) ;

assertEquals(c, p1.position+1);}

2.5 Développement dirigé par les Tests : TDD (Test Driven Development)

Principe : écrire notre méthode de test AVANT d'écrire la méthode à tester. Eclipse à un grand support pour TDD.

Application : Utiliser « first test approach » pour écrire la méthode toString de la classe Etudiant

public void testToString() {

Etudiant e3= **new Etudiant ();**

e3.setNom("aaaa bbbb");

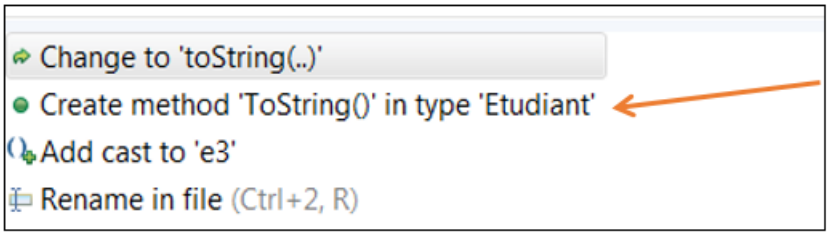
e3.setPrenom("cccc ");

e3.setAge(40);

String phrase ="mohamed el amine (age = 63)";

assertEquals(phrase, e3.ToString());}

- **Le compilateur demande d'ajouter la méthode « *ToString* » :**



Bénéfices du TDD

- Forcer à créer une spécification détaillée avant de code.

- Une méthode de test sert à une spécification, une documentation et une unité de test.

- Quand une méthode passe le teste, donc la méthode est correcte et complète.

Université de Mohamed Khider – Biskra –
Département d'informatique

Niveau : 1ère Master (GLSD) **Module : TTL**
Année : 2023

TP N° 1:

Application de l'outil JUnit sous éclipse

- Construire un programme OO en java de votre choix.
- Créer les classes de test sous JUnit.
- Afin de vérifier certaines méthodes du code java.
- Proposer des jeux de test les plus pertinents.
- Appliquer toutes les assertions possibles.
- Corriger votre code suivant vos corrections.
- Rendre un compte rendu sur votre travail.

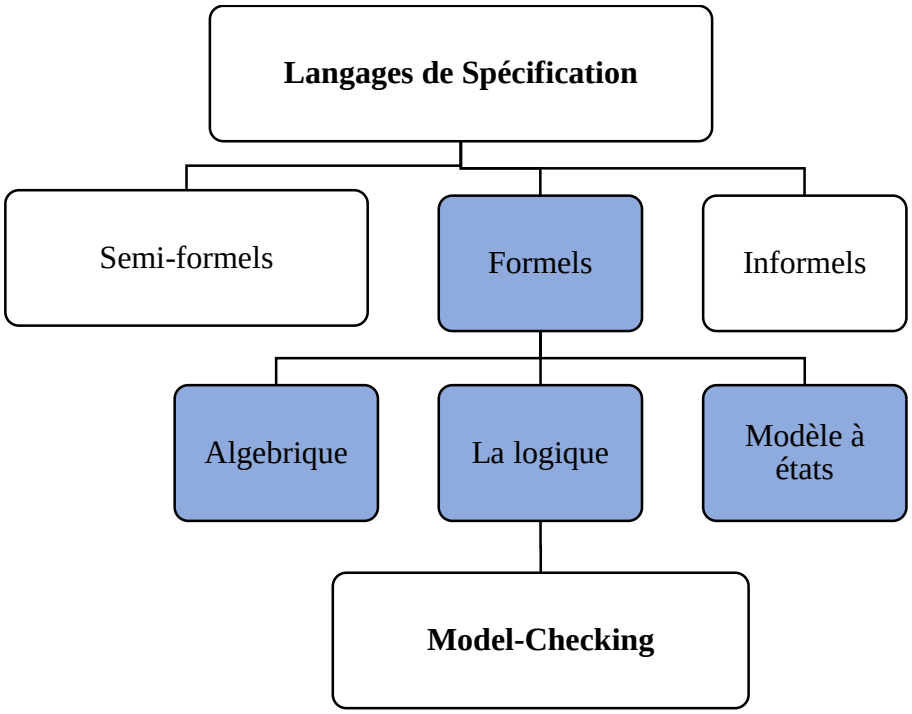
Chapitre 4 :

Initiation aux techniques de vérification formelle

1. Introduction

La **spécification** de certain système se ramène à l'étude des relations qui existent entre plusieurs composants caractérisant le système étudié. La **modélisation** est une activité technique qui s'inscrit dans de nombreux processus d'ingénierie, son but est de fournir une **représentation** approchée du système ou du produit que l'on veut analyser, concevoir ou fabriquer. De cet effet, il existe dans la littérature des **catégories** des spécifications nous allons étudier : la spécification semi formelle tel que UML, la spécification formel tel que les RDP, les automates d'états finis et des langages formel tel que la logique temporel.

2. Types des approches de vérification formelle



2.1 Technique de vérification formelle :

Les méthodes formelles sont des techniques alternatives, fondées sur des bases athématiques, permettant : La spécification. Le développement de systèmes. La vérification automatique de propriétés.

2.1.1 Logique de Hoare : Une méthode formelle définie par le chercheur en informatique britannique Tony Hoare en 1969. La logique Hoare a des axiomes pour toutes les instructions (si , sinon ,boucle .. Etc . Même des règles pour les procédures, les sauts, les pointeurs. ***Le triplet Hoare :***

{P} i {Q}

P : pré-condition i: exécution **Q**:post-condition

Exemple : Un exemple valide :{x>0} x :=x+1 {x>0} // car (x>0)+1 est tjrs >0

Un exemple non-valide :{x>0} x :=x-1 {x<2} //car (x>0)-1 n'est pas tjrs <2

Avantage : La méthode met en place un formalisme logique permettant de raisonner sur la correction des programmes informatiques.

Critique *:* -Le système de Hoare est complexe à utiliser car: Règles lourdes à appliquer. Taille des arbres de preuve. La logique de Hoare permet de démontrer qu'un programme vérifie une spécification.

2.1.2 Spécification algébrique

Les spécifications algébriques sont une technique permettant de définir le comportement (c'est-à-dire spécifier) des logiciels au travers des structures de données qu'ils manipulent (les listes, les files, les piles, les arbres, etc.). La manière la plus commune de caractériser une spécification SP est de donner un ensemble d'axiomes Ax composé de formules de la logique des prédicats du premier ordre.

On note alors SP = (Σ, Ax) et on dit que SP est une spécification axiomatique.

Exemple : les piles Soit une pile : Les opérations vide et empiler sont les constructeurs de pile. L'opération dépiler permet d'enlever le dernier élément empilé dans une pile. L'opération haut renvoie ce dernier élément sans le supprimer. L'opération hauteur renvoie la hauteur de la pile (le nombre d'éléments).

Soit ΣPiles = (SPiles, FPiles, VPiles) une signature équationnelle telle que :

SPiles = {entier, pile}

FPiles = {0 : entier, succ : entier → entier,

vide :→ pile,

empiler : entier × pile → pile,

depiler : pile → pile,

haut : pile → entier,

hauteur : pile → entier}

VPiles = {x : entier, p : pile}

La spécification des piles **Piles** est composée de la signature ΣPiles et de l'ensemble AxPiles d'axiomes suivant :

Depiler (vide) = vide

depiler (empiler(x, p)) = p

haut (vide) = 0

haut (empiler(x, p)) = x

hauteur (vide) = 0

hauteur (empiler(x, p)) = succ (hauteur(p))

a- Limitations de la vérification formelle algébrique :

- Pas de notion absolue de correction.
- La correction est toujours relative à une spécification donnée.
- Difficile et coûteux d'écrire des spécifications formelles.
- En pratique, on ne spécifie_pas formellement toutes les fonctionnalités mais les propriétés de sûreté comme la bonne formation des données (accès hors des bornes, déréférencement du pointeur nul, etc.), absence d'exceptions non détectées, les parties critiques du logiciel etc.
- Coûteux également de faire des preuves (mais on gagne sur le temps de test).

2.1.3 Model à Etats

a- Les Machines à Etats Finis (FSM) **:** Une machine à états finis est un modèle **dynamique**, c'est à dire qui évolue dans le **temps**. Cette évolution se traduit par des **changements** d'états en fonction des **entrées** permettant d'activer les transitions. Ceci étant, l'exécution commence toujours par un état **initial**, puis, en fonction de l'activation des **transitions**, la machine **change** d'état. A noter qu'à tout instant le système doit se trouver dans un état **unique**. Les FSMs sont utilisés pour décrire le **comportement** du système sous test. Les différents chemins (séquences de transitions) dans les FSMs s'apparentent à des scenarios de **test**. Ainsi, les FSMs ont largement été utilisés comme modèles **d'entrée** pour **la génération automatique de cas de test.** FSM est généralement représenté sous forme graphique avec des états (nœuds) et des transitions (arcs). Typiquement un FSM est un **4-uplet (E, T, In, Out) où E** est un ensemble fini **d'états**. **T** est un ensemble fini de **transitions**. **In** est un ensemble **d'entrées** provoquant les transitions. Une entrée peut par exemple représenter une expression booléenne définie sur une transition pour spécifier les conditions de passage d'un état vers un autre. **Out** est un ensemble de **sorties** qui peuvent se produire lors des transitions. Par exemple, une sortie peut représenter une action provoquée par le passage d'un état vers un autre. La figure suivante montre la spécification d'un chronomètre par une MEF.

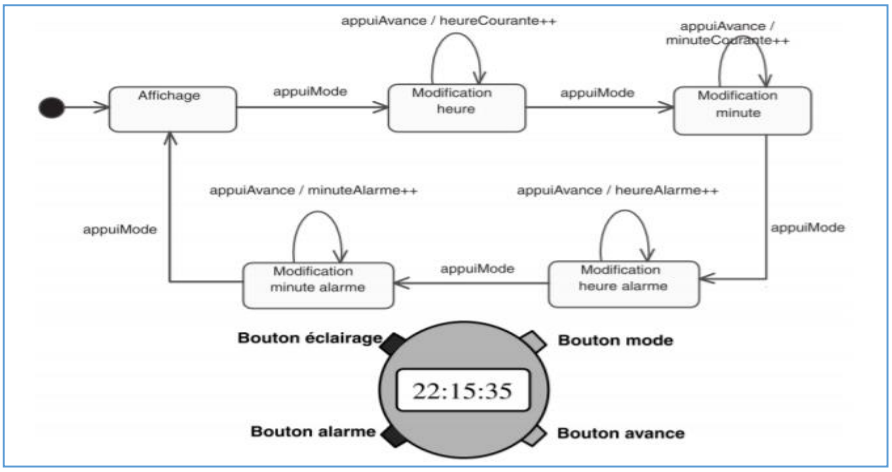


Figure 1: MEF d'un chronomètre

b- Réseaux de Pétri : Les réseaux de Pétri (Petri nets) inventés par Carl Adam Petri en 1962, sont une forme spéciale de graphe orienté possédant deux types de nœuds (repartis en deux groupes distincts) et des arcs reliant un type de nœud à l'autre. Typiquement, un réseau de Pétri est composé de trois groupes d'éléments : Des nœuds de **type P** appelés places qui représentent souvent les possibles états ou conditions nécessaires à l'accomplissement d'une action. Des nœuds de **type T** appelés transitions qui représentent souvent des événements ou des actions causant les changements d'états du système. Des **arcs** orientés avec chacun connectant une place à une transition, ou une transition à une place. On appellera entrées d'une transition tous les prédécesseurs de cette transition dans le graphe. De même tous les successeurs de cette transition dans le graphe seront appelés sorties. Par exemple, le réseau de Pétri de la Figure (a) comporte trois places p1, p2 et p3, une transition t1 et trois arcs (p1, t1), (p2, t1) et (t1, p3). Ainsi p1 et p2 représentent les entrées de t1, et p3 sa sortie.

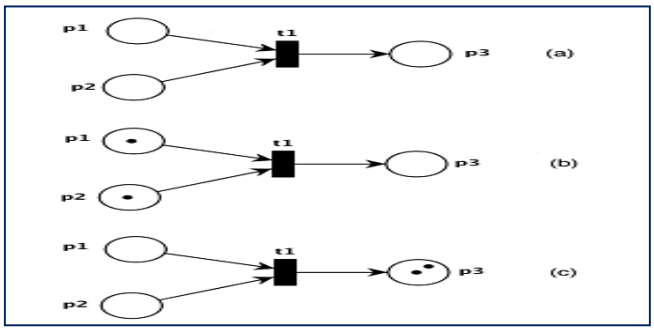


Figure 2 : RDP ordinaire

L'exécution d'un réseau de Pétri se traduit par des changements d'états. Un changement d'état est caractérisé par un mouvement d'un ou de plusieurs jetons, de places en places, en considérant le déclenchement d'une ou des transitions. Le déclenchement d'une transition peut représenter une occurrence d'un événement ou d'une action. Par ailleurs le déclenchement d'une transition ne peut se produire que si elle est au préalable active, c'est-à-dire que chacune de ses entrées contient au moins un jeton. Ainsi le déclenchement se traduit par le déplacement des jetons des entrées de la transition vers ses sorties.

Considérons maintenant le réseau de Pétri de la Figure (2 (a)), la présence d'un jeton dans chacune des entrées de t1 rend cette transition active. Ça n'aurait pas été le cas si p1 et/ou p2 n'avait pas de jeton. Le déclenchement de t1 produit le déplacement des jetons de p1 et p2 vers p3 (Figure 2 (c)). Il existe plusieurs variantes de réseaux de Pétri, on distingue entre autres : les réseaux de Pétri **colorés** (coloured Petri nets). Les réseaux de Pétri **temporels** (timed Petri nets).

Les réseaux de Pétri sont souvent utilisés pour décrire le comportement de systèmes **complexes**, en particulier les systèmes **distribués**. Ainsi, ils ont été utilisés comme **modèles d'entrée pour la génération automatique de cas de test.** Les réseaux de Pétri sont d'abord utilisés pour décrire le comportement des éléments du système **sous test**. Ensuite, l'approche propose des critères de **couverture** (par exemple, couverture des transitions du modèle). Ainsi, en fonction du critère choisi, un **arbre** de test est généré à partir du réseau de Pétri et les tests sont générés en parcourant l'arbre de la racine aux feuilles. Aussi, les réseaux de Pétri colorés (coloured Petri nets) ont été utilisés pour spécifier un système de **contrôle** de train. Ensuite à

partir du réseau de Pétri une approche permettant de **générer** des tests à partir de ce formalisme est présentée.

Les réseaux de Pétri offrent une notation puissante permettant de modéliser et/ou de simuler des systèmes complexes, par exemple, des systèmes distribués. Cependant, ils sont peu utilisés et peu connus dans le domaine de la génération de cas de test. La raison est que la notation est très complexe, peu intuitive et difficile à maintenir (par exemple s'il y a des modifications à apporter au modèle suite à une modification des spécifications).

c- **Modèles Stochastiques (chaines de Markov)** Les modèles stochastiques permettent de représenter un système en se basant sur des notions **probabilistes**. Les chaines de Markov sont un exemple typique des modèles stochastiques. Ils permettent globalement de représenter l'évolution d'un système dans le temps en utilisant des modèles de transitions associés à des probabilités (sur les transitions). L'une des propriétés importantes des chaines de Markov est **l'absence de mémoire**, c'est-à-dire que le prochain état du système ne dépend que de l'état courant de celui-ci et non des états passés. L'exemple de la Figure 3 présente une chaine de Markov (discrète) à deux états (A et B) et quatre transitions, chacune associée à une probabilité. Ainsi, à partir de l'état A, le système passe dans l'état B avec une probabilité de 2/3 ou reste dans l'état A avec une probabilité de 1/3. Dans l'état B le système passe dans l'état A avec une probabilité de 1/2 ou reste dans l'état B avec également une probabilité de 1/2. On remarquera que la somme des probabilités associées aux transitions sortantes d'un état quelconque vaut 1.

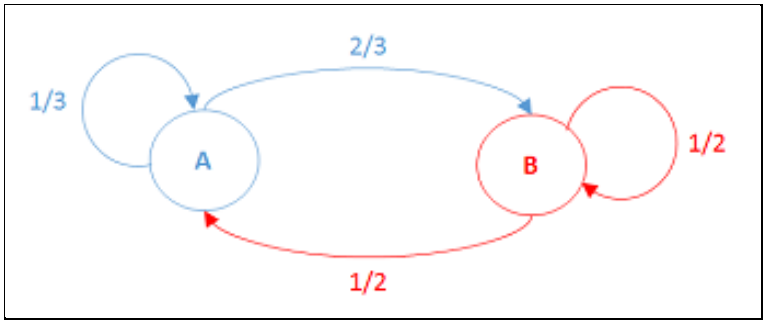


Figure 3 : exemple d'une chaine de markov.

d- **Model checking:** le Principe est que Un vérificateur de modèle prend le modèle extraites du système et les **propriétés** en entrée et recherche de manière **exhaustive** les erreurs éventuelles. Autrement dit, **vérifier si un système S satisfait une propriété P**. il est fortement appliqué dans: protocoles de communication (les précurseurs). Circuits électroniques. Algorithmes distribués. Programmes réactifs et temps-réel. Il s'agit d'un graphe : un sommet représente un état du système et chaque arc représente une transition. La logique temporelle est le langage le plus utilisé dans model cheking : Il existe plusieurs logiques temporelles : LTL (linear tomporal logic) et CTL* (computational tree logic) .. Etc..La propriété à vérifier est écrite par une formule de logique temporelle. Il suit les suivantes étapes (voir figure 4) **:**

- **Phase de modélisation :** Modéliser le système à l'étude. Formaliser la propriété à vérifier
- **Phase d'exécution :** Exécuter le model checker pour vérifier la validité de la propriété dans le modèle
- **Phase d'analyse :** Propriété satisfaite ?→ Vérifier la propriété suivante.

Propriété violée ? 1. analyser le contrexemple généré par simulation. 2. affiner le modèle, le design du modèle ou la propriété . . . et répéter la procédure pas assez de mémoire ? Essayer de réduire le modèle et tester à nouveau

Le modèle checking porte les qualités suivantes : **Fiabilité :** Assure la propriété de sureté qui assure la fiabilité. **Exhaustivité :** Vérification exhaustive. **Déploiement :** Durant le développement. **Coût :** Moins cher

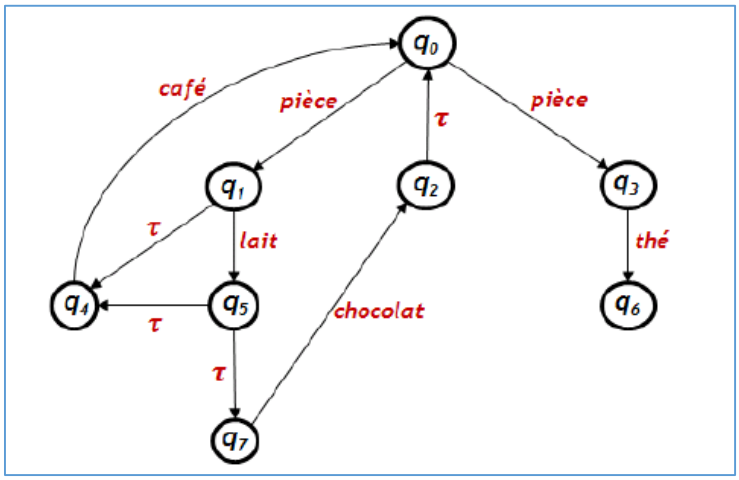


Figure 4 : Distributeur de café

e- Comparaison entre la Logique de Hoare et le Model checking

Critère	**Techniques**	
	Hoare logic	**Model-cheking**
Automatisation	- Non-automatisable	+ Basé sur l' automastisation
Preuves	conception	conception
Applications	+++ illlimité	+ limités
comprehension	- Difficiles	+ moyen
rapidité	+ Prend du temps car c'est des preuves ecrites a la main	+++ plus rapide par rapport à d'autres techniques
validation	++	+
Assisté par ordinateur	-	+

3. Application 1 : Modélisation formelle d'un bruleur à gaz par les AEF temporisés en utilisant UPPAAL

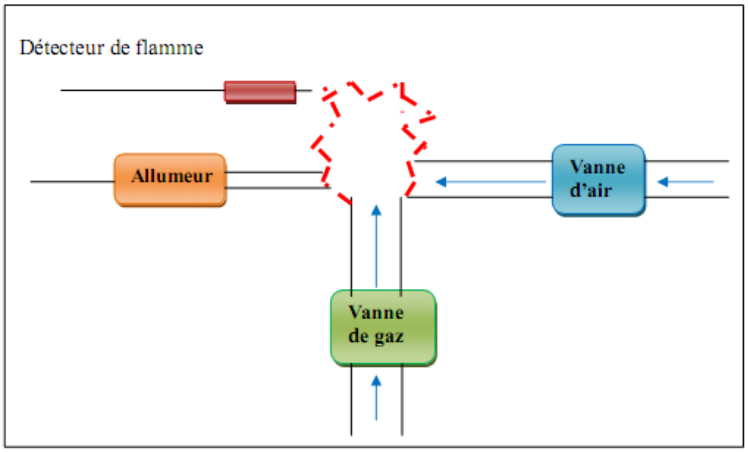


Figure 5 : Exemple de bruleur de gaz

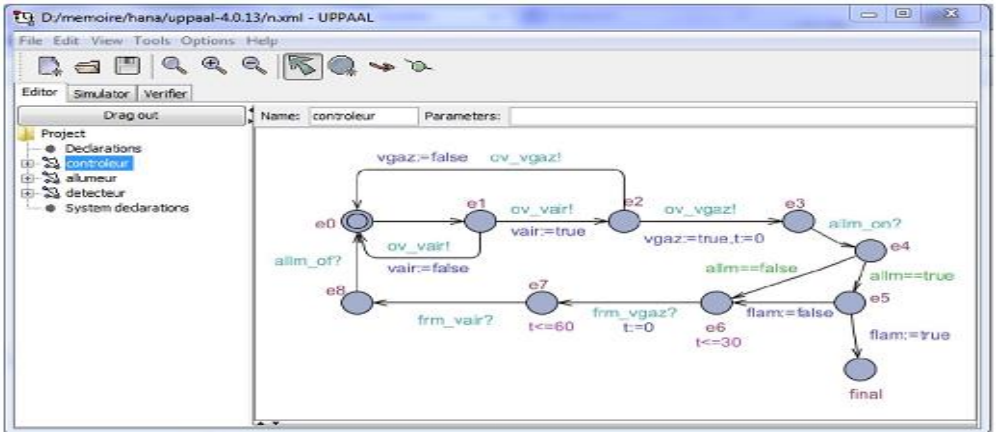


Figure 6 : Modèles de processus contrôleur

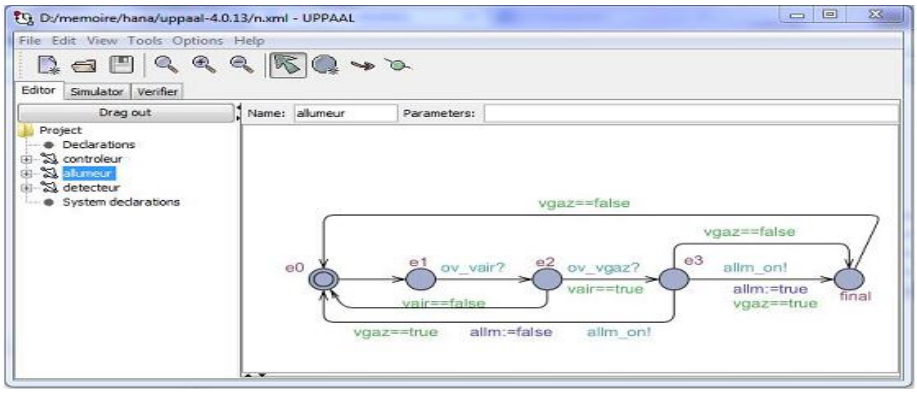


Figure7 : Modèle de processus allumeur

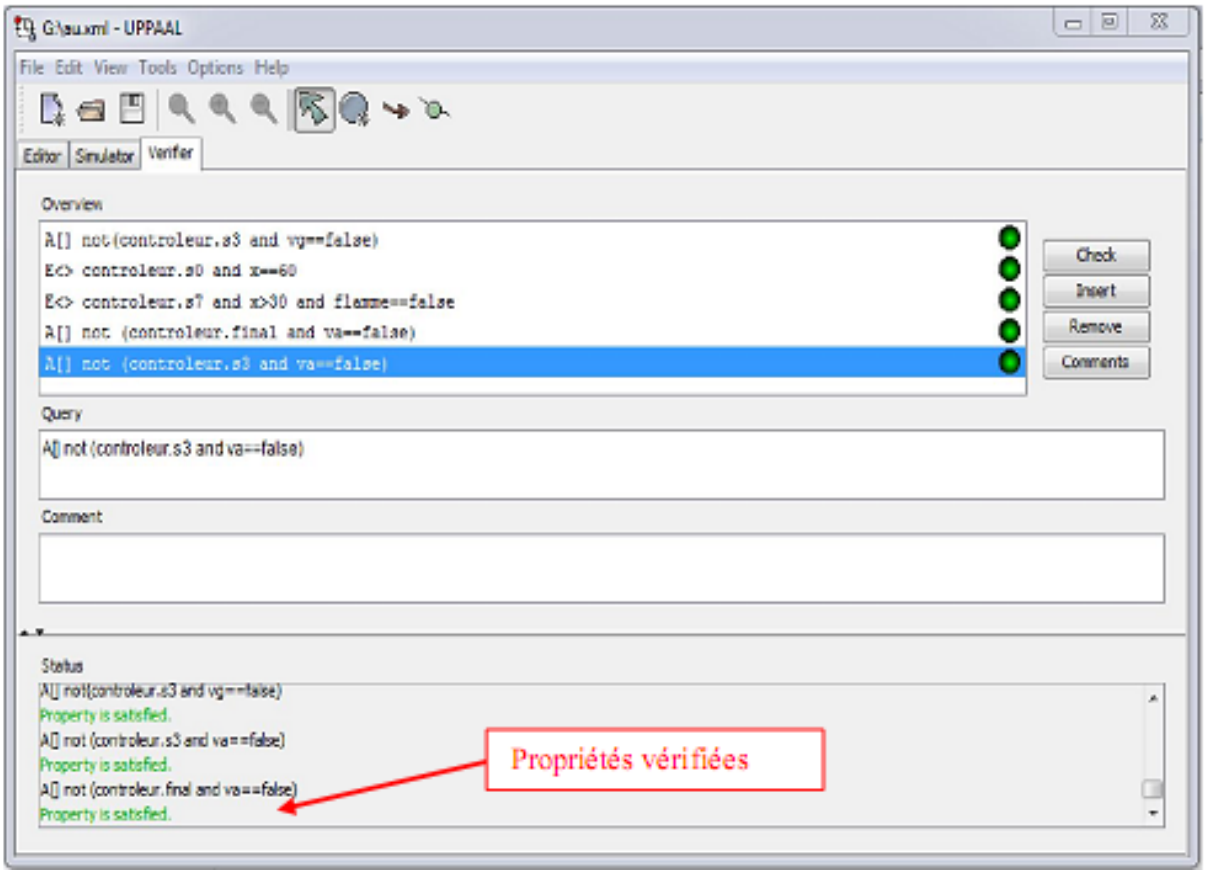


Figure 8 : Résultat de vérification formelle par l'UPPAAL

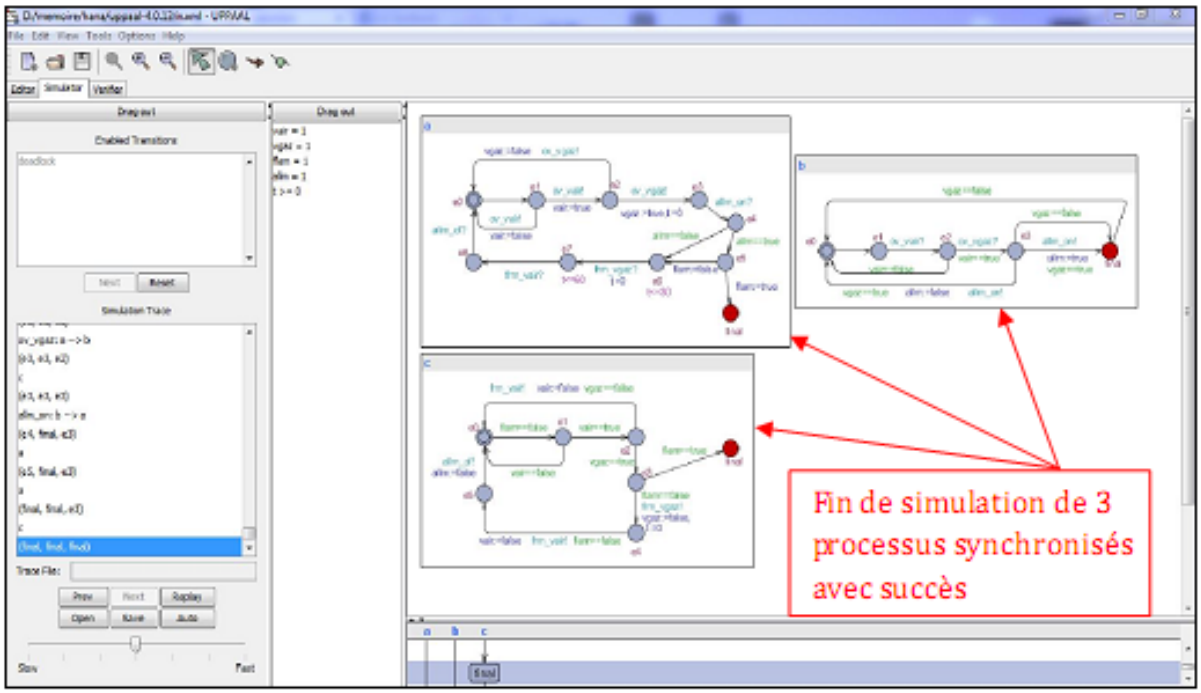


Figure 9 : Simulation de comportement synchronisé

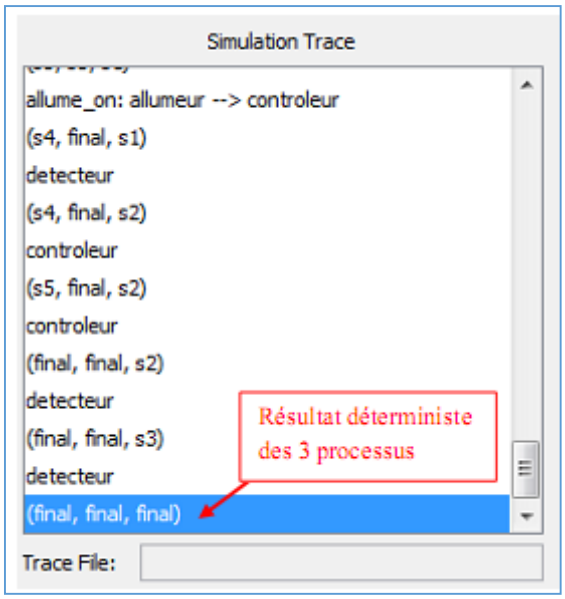


Figure 10 : Résultat de la simulation synchronisée

Conclusion **:** Les applications sont de plus en plus **complexes**, les **volumes** de données sont de plus en plus grands...Les tests sont aujourd'hui au centre de tous les intérêts : de nombreux **progiciels** ont vu le jour pour tester, gérer les versions… Des **sociétés** ont investi dans la création d'un service interne, véritable structure de tests. Rien ne peut être mis en production sans être validé par ce service. L'Internet mobile, nouveau mode d'information et nouveau challenge pour les entreprises, est un support graphique différent nécessitant une programmation adaptée et donc des tests complémentaires.

Université de Mohamed Khider – Biskra –

Département d'informatique

Niveau : 1ère Master (GLSD) **Module : TTL**

Année : 2023

TD N°4

Exposé : faire ton propre étude comparative entre les différentes techniques de vérification formelle, en respectant le tableau suivant :

Paradigme formel	Format : texte/ graphe/..	Outils logiciel	Domaine	Coût	Quand	Qui	Remarque
RDP							
AEF							
Hoare logic							
Model-cheking							
chaines de Markov							
Spécification algébrique							
Logique de Hoare							

Références bibliographiques

- Glenford J. Myers, Tom Badgett, Corey Sandler , “The Art of Software testing”, deuxième Edition, Published by John Wiley & Sons, Inc., Hoboken, New Jersey, 2012.
- Cours 1: Prof. Youness Boukouchi, les fondamentaux du test logiciel, ENSA d'Agadir, 2017.
- Cours 2: Stefano Zacchiroli, Génie Logiciel Avancé – Introduction, Paris7, 2011.
- Cours 3: Vérification & Validation, Philippe Collet, université de Nice, 2008.
- Thèse : SekouKangoye, Elaboration d’une approche de vérification et de validation de logiciel embarqué automobile, basée sur la génération automatique de cas de test. Université d’Angers, 2017.
- Larman C., UML 2 et les design patterns. Troisième édition. Pearson Éducation. Chapitre 16, 2005.
- Oduar Mejia Lopez. “Visualisation de la cohésion et du couplage du code Java”, UNIVERSITÉ DE SHERBROOKE Faculté de génie Département de génie électrique et de génie informatique, Mémoire de maîtrise Spécialité : génie électrique. 2011.