Chapitre 4 Test logiciel

1. INTRODUCTION

La validation d'un programme est un processus continu qui s'étend sur l'ensemble de son cycle de vie. La preuve de la correction des programmes est difficile de mise en oeuvre et est, généralement, limitée aux programmes de petite taille. Les tests constituent, en pratique, la méthode de validation la plus largement mise en oeuvre dans la plus part des organisations.

Il est très important de comprendre que les tests ne permettent pas d'établir qu'un programme est correct. Des erreurs peuvent passer inaperçues, même après la mise en oeuvre de tests extrêmement sophistiqués. Les tests permettent de démontrer la présence d'erreurs dans un programme, pas leur absence. Suivant Myers , un test est bien conçu s'il permet d'établir la présence d'une ou plusieurs erreurs.

D'un autre côté, la connaissance détaillée de la structure d'un logiciel peut être extrêmement utile pour construire des tests judicieux. Le rôle du programmeur peut donc être très important. Il convient donc d'établir un environnement de travail associant les programmeurs et des personnes extérieures à la réalisation du système.

Le test n'est rien d'autre que la vérification de la cohérence entre un programme et sa spécification au moyen d'un échantillon de données. Comme, en général, un programme ne peut être exécuté pour toutes les données d'entrées possibles, on ne peut qu'utiliser un sous-ensemble pertinent (échantillon) des données d'entrée. Sous-ensemble généralement beaucoup plus petit que l'ensemble possible (réel) des données d'entrée. Par essence, le test n'est qu'un échantillonnage qui repose sur l'hypothèse implicite que si le programme testé fournit des résultats corrects pour l'échantillon choisi, il fournira aussi des résultats corrects pour toutes les autres données. C'est là le point faible du test sa validité dépend entièrement de la pertinence de l'échantillon utilisé, c'est-à-dire de celle du sous-ensemble des données choisi.

La question clé est donc comment choisir l'échantillon en entrée et comment évaluer son efficacité.

Un autre problème de base réside dans le fait qu'un programme ne peut être testé

en une seule fois. Car les opérations de test et de mise au point qui en résultent seraient alors beaucoup trop complexes. En conséquence, le principe « diviser pour régner» doit être appliqué, c'est-à-dire crue les opérations de test, à l'instar des opérations de conception, sont découpées en plusieurs étapes.

L'objectif du test est de mettre en oeuvre le logiciel en utilisant des données similaires aux données réelles, pour observer les résultats, détecter des anomalies (défaillances), et en déduire l'existence d'erreurs, donc de fautes. Le test est un processus qui a pour objectif d'exécuter un programme avec l'intention de trouver des erreurs dans le but d'apporter une valeur ajoutée en matière de qualité et fiabilité. Le coût de ce processus doit être compensé par l'amélioration de la valeur du produit logiciel testé en terme (de fiabilité et qualité).

"Le test est l'exécution ou l'évaluation d'un programme et de ses données, par des moyens automatiques ou manuels, pour vérifier qu'il répond a ses spécifications ou pour identifier les différences entre les résultats attendus et les résultats obtenus.

2. MODELISATION ET REPRESENTATION D'UN PROGRAMME

Tester un programme revient à trouver des données similaires aux données réelles manipulées par le système. On doit vérifier que des données correctes choisies n'ont pas d'influence sur l'intégrité des autres composants du système.

Il existe, principalement, deux manières de modéliser un programme. Les cas de test son construits selon le modèle choisi. Ces deux manières sont résumées dans ce qui suit:

1) Test fonctionnel : Les cas de test sont construits â partir des spécifications. Dans ce cas, on s'intéresse aux fonctionnalités du programme. Le programme est vu comme une boite noire.

2) Test structurel : Les cas de test sont construits à partir de la structure interne du programme. Le programme est vu comme une boite ouverte ou boite blanche.

Dans les deux approches, les cas de test d'un programme doivent comporter les informations

suivantes:

* une spécification des données d'entrée.
* une spécification des données attendues en sortie (résultats),

* une description des fonctions du système qui sont testées.

3. LES TECHNIQUES DE TEST

3.1. Introduction

Il semble que les spécialistes des techniques du test se répartissent en deux écoles qui communiquent peu entre elles : d'une part, les tenants du test statique et des modèles de fiabilité et, d'autre part, les tenants des techniques de jeux de données de test. La deuxième

école peut elle même être subdivisée en deux. En premier lieu, les partisans du test dit de « boite noire» ou test fonctionnel. L'expression boite fermée ou boite noire, provient du fait que l'élément testé, qui peut être par exemple un module, n'est considéré qu'au travers de ses interfaces (données d'entrée, données de sortie), en ignorant tout de la structure de sa réalisation. Dans un second lieu il y a les partisans du test de « boite blanche » ou test structurel. 1.'expression boite ouverte ou boite blanche, provient du fait que les tests structurels prennent en compte la structure de contrôle du programme et parfois la structure de ses données. Les données sont alors choisies pour suivre, des chemins particuliers au travers ces structures.

3.2. Les tests fonctionnels (boite fermée)

3.2.1. Principe

Un programme est composé d'un ensemble d'unités et chaque unité doit réaliser une fonction bien définie. Le programme à valider est exécuté avec des jeux d'essai préalablement établis et on s'assure que ce que fait le programme est conforme à ce qui a été défini dans un document de référence. Le test fonctionnel permet de répondre à la question: « est ce que le programme fait ce qu'il veut qu'il fasse » . 1a figure 1 résume ce principe :

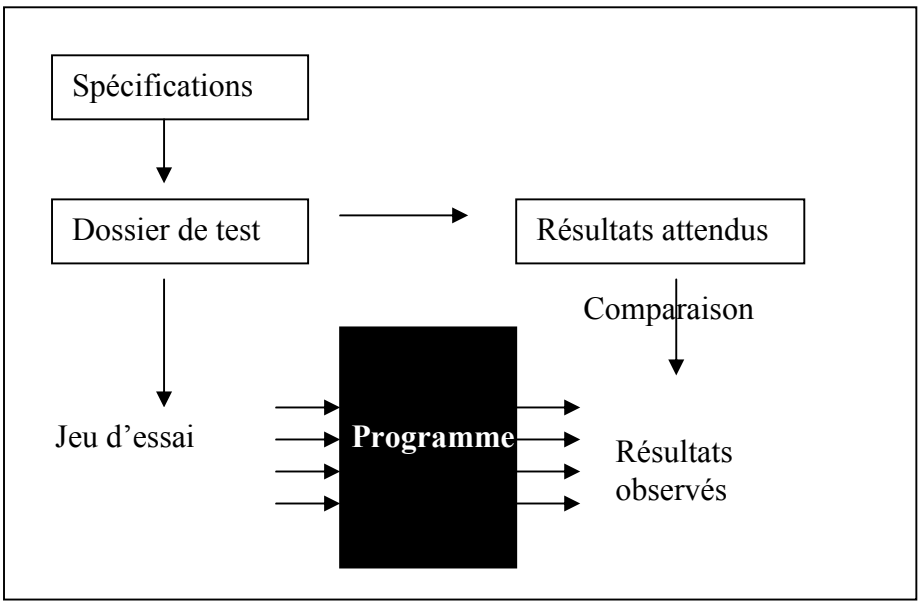


Figure 1 : Principe du test fonctionnel

Les principales techniques de base utilisées pour les tests fonctionnels sont

* le test par les classes d'équivalence,
* le test aux limites,
* le test par relation 'cause-effet',
* le test par estimation d'erreurs

3.2.2. Le test par les classes d'équivalence

Pendant le test du programme, il sera préférable d'identifier des classes d'équivalence vis-à-vis d'une condition 'externe' à l'élément testé. L'objectif principal de cette technique est de contrôler que les données en sortie (résultats observes), correspondent aux différentes classes d'équivalence définies préalablement, ci sont conformes aux spécifications.

Le test par classes d'équivalence doit se limiter, souvent, à un sous ensemble de données d'entrée possible, qui a une grande probabilité de permettre la découverte d'erreurs (fautes). La conception des cas de test se fait en deux étapes :

* Identification des classes d'équivalence on doit les identifier à partir des

spécifications et prévoir des classes d'équivalence pour. les entrées valides et des classes d'équivalence pour les entrées invalides.

***** Définition des cas de test les classes d'équivalence identifiées précédemment sont utilisées pour construire les cas de test du programme.

Les cas de test choisis doivent parcourir les classes identifiées.

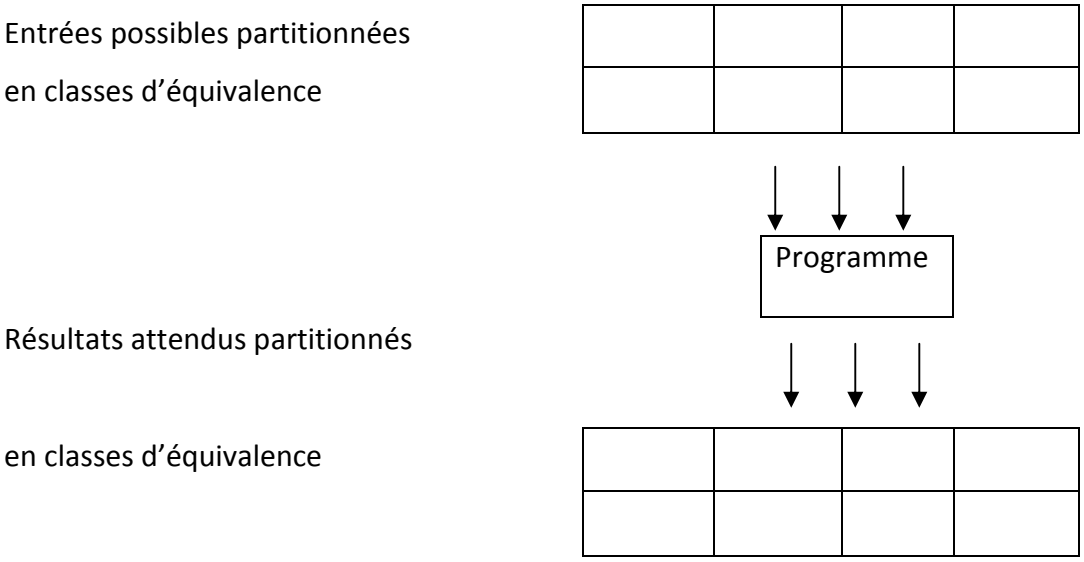


Figure 2 Principe du test par les classes d'équivalence.

3.2.3. Le test aux limites

On divisant les entrées et les sorties en classes d'équivalence, il est important de s'intéresser aux valeurs limites de ces classes valeurs, Dans ce cas les cas de test sont construits pour les données égales ou proches des bornes des classes d'équivalence.

3.2.4. Le test par relations 'causes / effets'

L'une des faiblesses du test aux limites et du test par classes d'équivalence est qu'ils ne prennent pas en considération la combinaison des données en entrée. Cette technique permet de compléter les approches précédentes. En effet, il apporte une aide pour le choix des combinaisons de données d'entrée choisies dans les classes d'équivalence. Il permet également de détecter des erreurs dans les spécifications, il

consiste à représenter les spécifications sous la forme d'un graphe 'cause-effet', puis à traduire ce graphe sous la forme d'une table de décision dont les colonnes représentent les tests à effectuer. Le principe de cette technique est illustré par la figure 3.

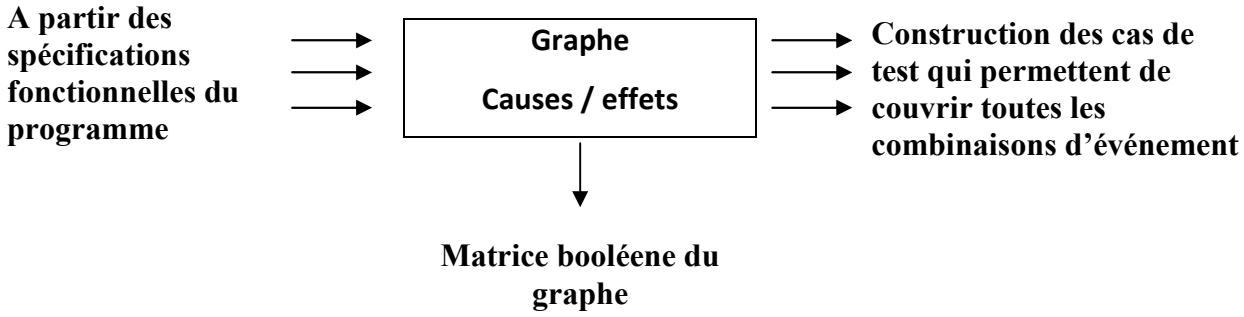


Figure 3 : Le test par relation 'causes/ effets'

3.2.5. Le test par estimation d'erreurs

Il est difficile d'établir une procédure pour l'estimation des erreurs vu que le processus est intuitif. L'idée de base est d'énumérer une liste d'erreurs possibles puis construire les cas de test nécessaires pour arriver à détecter ces erreurs.

3.2.6. Conclusion

L'inconvénient du test fonctionnel, est qu'il ignore quelques propriétés fonctionnelles des programmes qui sont une partie de la conception (implémentation), et qui ne sont pas décrites dans les spécifications.

Les différentes techniques du test fonctionnel présentées dans ce paragraphe, peuvent être combinées pour la construction des cas de test. Un effet, chacune d'elles peut contribuer à la construction d'un ensemble particulier ce cas de test.

3.3. Les tests structurels (boite ouverte)

3.3.1. Principe

Les tests structurels sont fondés sur les structures internes du programme à tester. Le programme à valider est exécuté avec des jeux d'essai préalablement établis et on s'assure que la structure interne du programme est « bien » parcourue par les jeux d'essai. Le test structurel permet de répondre à la question « est-que toutes les

parties du code clans le programme ont été exercées ». Le principe de ce type de test est illustré par la figure 4.

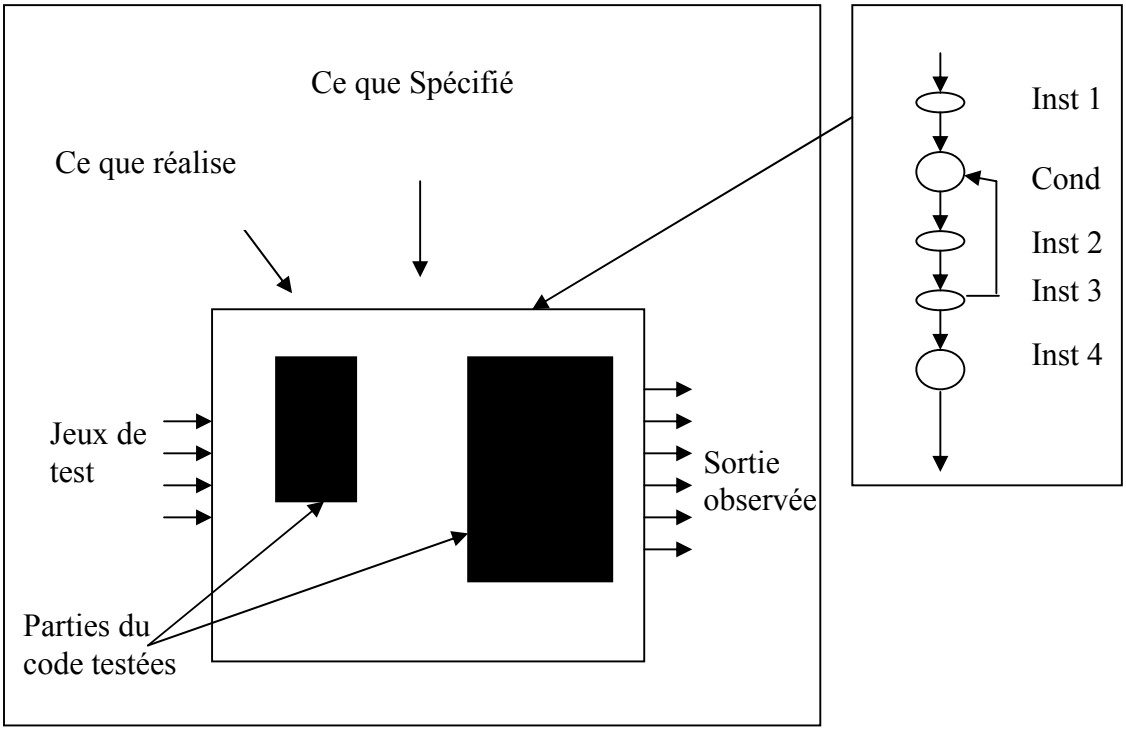


Figure 4 : Principe du test structurel

Pour effectuer le test structurel, on peut procéder comme suit:

- Etablir les graphes de contrôle,
- Décomposer statiquement le programme à tester en portion de code **»,**
- Exécuter le programme avec des jeux d'essai préétablis,
- Marquer « les portions de code » exercées afin (le mesurer une couverture de test.

3.3.2. Définitions

Dans ce type de test, on s'intéresse à la structure interne du programme (boite ouverte). Elle sera représentée sous forme de graphe de contrôle et permet de faciliter la tâche, lors du test, par marquage des différentes structures parcourues. Nous donnons, dans ce qui suit, la définition des différents éléments composants la structure interne des programmes.

Branche:

Une branche est une suite d'instructions, menant d'un noeud vers un autre du graphe, telles que l'exécution de la première instruction de la branche entraîne celle de la dernière.

Chemin:

Un chemin est une succession de branches entre un point d'entrée et un point de sortie du programme. Il existe différents types de chemin

Chemin de contrôle:

Un chemin de contrôle est une séquence de noeuds du graphe de contrôle partant de l'entrée du programme et finissant à la sortie du programme.

Chemin d'exécution:

Un chemin d'exécution est un chemin de contrôle effectivement parcouru durant l'exécution du programme. Tous les chemins de contrôle ne sont pas des chemins d'exécution.

3.3.3. Types de tests structurels

On distingue les différents types de tests structurels suivants :

Test d'instructions : Ce type de test consiste a exécuter, au moins une fois, l'ensemble des instructions d'un programme.

Test de branches : Il consiste à exécuter, au moins une fois, le plus grand nombre de branches. Ce type de test sera pratiqué à deux niveaux différents.

*** Niveau composant:**

Il consiste à passer, au moins une fois, par toutes les branches du composant, qui ne sont autre qu'une séquence d'instructions comprise entre deux aiguillages du composant.

*** Niveau Programme:**

Ce type de test permet de vérifier la communication entre les différents modules d'un programme. Le principe est d'exécuter au moins une fois chaque appel de composant.

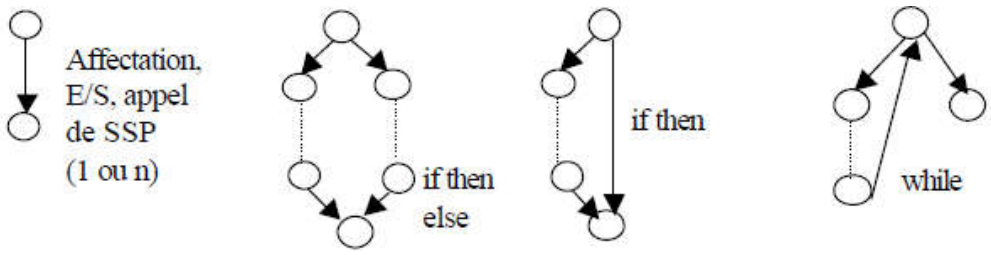
Le test de branche permet de contrôler et de s'assurer que toutes les branches ont été parcourues au moins une fois.

Test de conditions : Dans ce type de test, on s'intéresse aux conditions. Le principe

de ce type de test est de créer des cas de test à fournir en entrée de la condition et s'assurer que toutes les conditions sont exécutées au moins une fois. il faut choisir les différentes valeurs des variables permettant de valider les conditions qui sont parfois multiples.

Test de chemins : Le principe de ce type de test est d'exécuter le plus grand nombre de chemins possibles (chemins tirés du graphe de contrôle). Il nous permet d'identifier les chemins non exécutables (chemins qui ne sont jamais parcourus). Il constitue, en théorie, le type de test le plus efficace. Cependant, en pratique, le nombre total de chemins possibles dans un graphe de contrôle d'un programme rend ce type de test impraticable pour les logiciels de taille et de complexité importante.

Éléments d'un graphe de contrôle



3.3.4. Conclusion

Malgré sont efficacité, le test structurel présente quelque limites :

***** il est incapable, en lui même, de démontrer que des « parties de code » sont manquantes instructions, branches, ..., etc. entraînant des chemins d'exécution manquants.

* Par ailleurs, il ne peut pas détecter certaines erreurs sur les données. Cependant, les tests structurels constituent la technique la plus largement utilisée. Un de leurs avantages est qu'on peut (relativement facilement) développer des outils permettant de mesurer quantitativement l'efficacité des cas de test par les taux de couverture.

Pour les obtenir, on instrumente le code, c'est-à-dire on définit des points d'observation particuliers permettant d'étudier le comportement du programme lors de son exécution. Cette instrumentation s'effectue généralement par l'insertion d'instructions dans le code du programme testé.

3.4. Couverture de test

Définition : Une couverture de test est le rapport entre le nombre de tests effectués et le nombre de tests nécessaires pour que soit vérifiée une certaine propriété du logiciel testé.

Il existe deux types de couvertures de test :

*** Couverture fonctionnelle :** Dans ce cas on s'intéresse à la couverture des entrées et des sorties du logiciel.

*** Couverture structurelle** : Cm s'intéresse, clans ce cas, aux taux de couverture d'éléments de la structure interne du programme, tels que les instructions, les branches les décisions simples, multiples, les chemins, …etc.

4. LES ETAPES DU TEST

4.1. Introduction

De la même façon que peur les processus de conception et de programmation, les tests se feront par étapes. Chaque étape constituant la suite logique de l'étape précédente. On distingue plusieurs étapes dans ce processus.

4.2. Les tests unitaires

Au cours du test unitaire, un module est testé pour la première fois. La cohérence entre le code source et la conception du module est vérifiée. Les aspects considérés sont les algorithmes et l'utilisation des données. Le but principal est de vérifier la correction des algorithmes. Les tests unitaires présentent plusieurs avantages, parmi lesquels on peut citer:

***** concentration de l'effort de test sur chaque composant tu logiciel pris isolément, ce qui représente une tâche relativement simple, sauf si ce composant est fortement connecté aux autres,

* en cas de comportement anormal du composant, il est relativement plus facile de découvrir l'erreur, de la localiser, et enfin de la corriger.

* enfin, les tests unitaires introduisent la notion de parallélisme durant la campagne de test, dans le sens où plusieurs composants du logiciel peuvent être testés indépendamment des autres et surtout simultanément, ce qui permet de gagner du temps.

4.3. Les tests d'intégration

Les tests d'intégration supposent que chaque composant à intégrer a déjà été vérifié par un test unitaire. En fait, ce sont les aspects non encore testés qui doivent être considérés. En conséquence, à ce stade, les différents composants du logiciel qui ont été testés individuellement, vont être progressivement assemblés, pour construire un programme exécutable qui va être testé. Ses tests permettent de vérifier la bonne utilisation des interfaces entre modules et de vérifier le hon enchaînement durant l'exécution, entre les différents composants du logiciel.

4.4. Les tests de validation

Au cours des tests de validation, on procède à la vérification de la cohérence entre le programme et la définition du système qui décrit, les fonctions internes et externes, les structures de données et les états du système. Il permettent de tester le système, dans sa totalité, avec des données réelles. Ils mettent souvent en évidence des erreurs dans un logiciel ne satisfaisant pas les niveaux de fonctionnalité ou de performance attendus par l'utilisateur.

4.5. Conclusion

Les étapes du test décrites ci-dessus, doivent être planifiées à l'avance, relativement tôt dans le cycle de développement du logiciel (Ces étapes seront combinées selon les stratégies de test adoptées.

5. LES STRATEGIES DE TEST

5.1. Introduction

L'ordre dans lequel les tests sont effectués peut avoir des incidences sur :

***** la construction des cas de test,

***** la détection et correction d'éventuelles erreurs,

***** et éventuellement le choix des outils de test qui peuvent être utilisés.

On distingue, principalement, deux types de stratégies : la première approche est

appelée “non incrémentale”, tandis que la seconde est connue sous le nom d’approche “incrémentale”.

5.2. Les stratégies non-incrémentales

Les stratégies non-incrémentales consistent à tester de façon indépendante tous les composants du logiciel, puis à les intégrer (en un seul bloc). Cette approche implique, pour chaque composant à tester, la réalisation d’un moniteur de test (driver), et de simulateurs de composant (stub) pour chaque composant appelé. C’est donc une méthode coûteuse et peu utilisée. Une illustration de cette approche est donnée par la figure *5.*

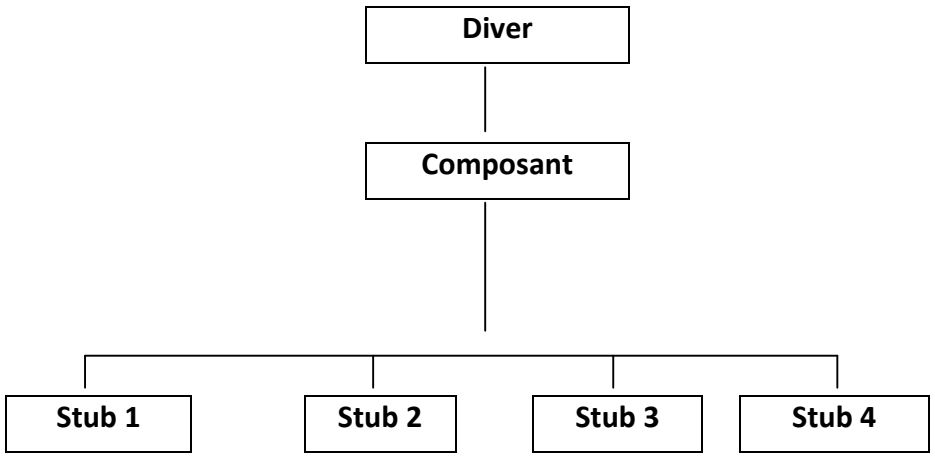


Figure 5 Stratégies non-incrémentales

5.3. Les stratégies incrémentales

5.3.1. Principe

Les stratégies incrémentales consistent à ne pas tester les composants isolément, mais les intégrer aux composants déjà testés, de manière progressive. Elles présentent l’avantage de limiter le nombre de moniteurs de test et de simulateurs nécessaires, dans le cas de cette approche, il est possible d’effectuer l’intégration de façon ascendante ou descendante, vis-à-vis de l’arbre de décomposition du logiciel. Une illustration de ces deux approches est donnée par la figure 6:

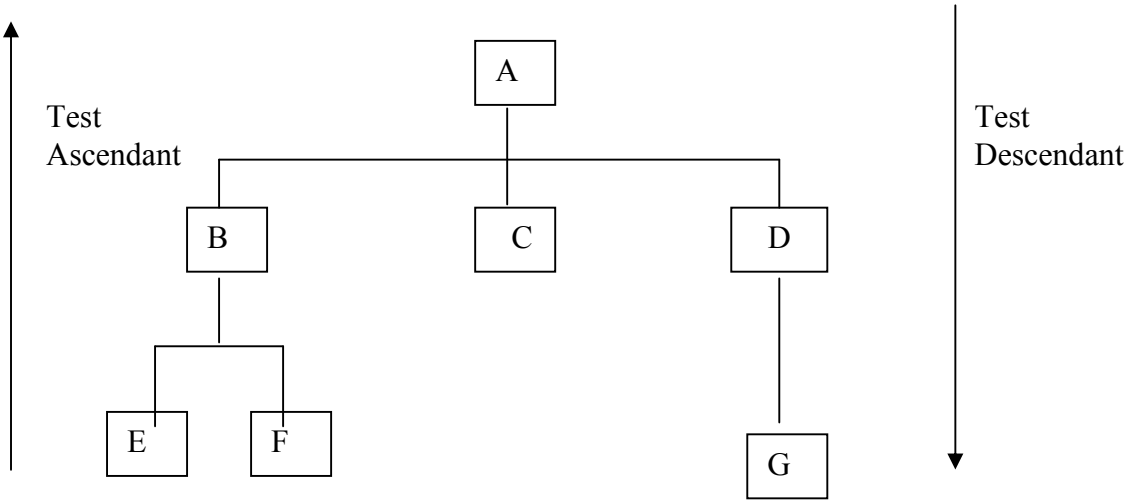


Figure 6 : Les stratégies incrémentales

5.3.2. Intégration par la démarche descendante

La méthode des tests descendants est généralement associée au processus de développement de programme descendant. Un module est testé dés qu'il à été codé. Avec la méthode descendante, on dispose rapidement d'une maquette du système, mais très limitée dans ses fonctions. Dans cette approche, les composants de plus haut niveau sont testés en premier. Les tests descendants peuvent être extrêmement coûteux parce que des versions provisoires de chaque fonction doivent simuler les couches basses du logiciel. Le principe de cette approche est illustré par la figure 7

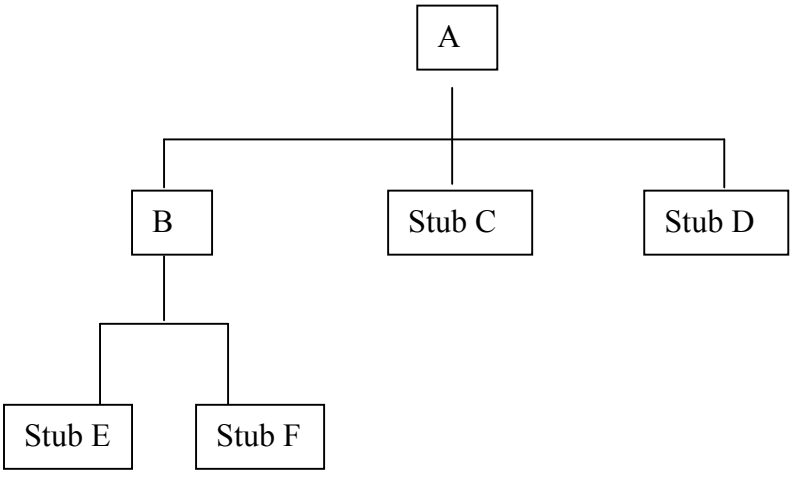


Figure 7 : Les tests descendants.

5.3.3. Intégration par la démarche ascendante

Les tests ascendants commencent au niveau d'un sous système avec des modules provisoires, qui ont la même interface que les modules définies mais qui sont beaucoup plus simples. Après avoir effectué les tests des sous systèmes, chaque module est testé de la même façon, en utilisant des fonctions provisoires. Puis les fonctions sont remplacées par le véritable code et celui-ci est testé. Pour réaliser des tests ascendants, Il faudra réaliser des programmes provisoires qui permettent d'appeler les modules à tester. Il sera alors plus facile d'observer les résu1tats du test. Une illustration de cette approche est donnée par la figure 8

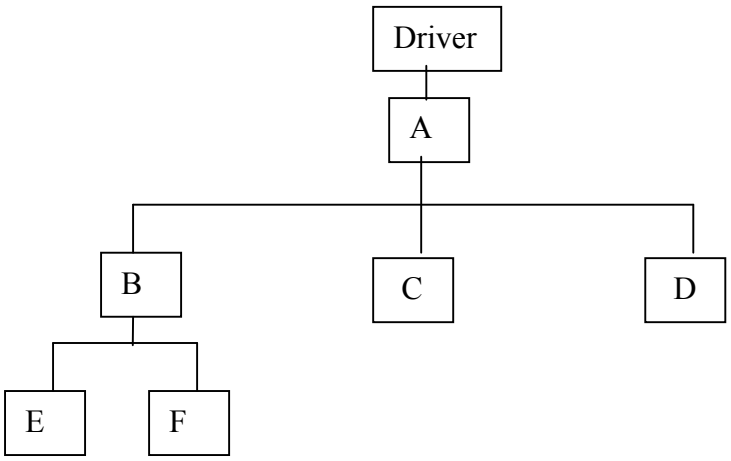


Figure 8 . Les tests ascendants

6. LES OUTILS DE TEST

6.1. Introduction

Le processus de validation d'un logiciel est laborieux et constitue souvent l'étape la plus coûteuse du cycle de vie. On a donc tout intérêt à utiliser des outils permettant, lorsque cela est possible, d'automatiser ou aider cette phase. Nous donnons, dans ce qui suit, une typologie des différents outils qui peuvent aider le processus de test.

6.2. Les moniteurs de test et simulateurs de composants

Ce type d'outils apporte une aide à la mise en oeuvre des composants testés et à la simulation des composants appelés. Des exemples de tels outils, génèrent des

moniteurs de test à partir des spécifications de packages ADA.

6.3. Les évaluateurs de taux de couverture de test

Ce sont des programmes qui permettent de savoir combien de fois, par exemple, chaque instruction d'un programme a été exécutée. Ils sont aussi appelés analyseurs dynamiques ou analyseurs de flot d'exécution.

6.4. Les simulateurs d'environnements

Un simulateur est un programme qui imite les actions d'un autre programme ou d'un dispositif matériel. Les simulateurs sont très souvent utilisés au moment des tests pour remplacer un matériel indisponible. Dans certains cas, c'est la seule façon de procéder pour certains tests. Ils sont généralement spécifiques de l'application. Ce test est utilisé lorsque le test dans l'environnement est impraticable ou revient trop chère.

6.5. Les gestionnaires de tests automatiques

Ce type d'outils permet de mémoriser toutes les informations relatives à un test, permettant ainsi de ré exécuter, soit pour effectuer une remise en condition d'erreurs, soit pour effectuer des tests de non-régression.

6.6. Les générateurs de jeux de test

Les générateurs de tests génèrent automatiquement de grand nombre de jeux de données d'entrée pour les tests de certains systèmes. Ils sont particulièrement utiles lorsque les performances du système doivent être testées dans un environnement réel.

6.7. Les modèles prédictifs

Ce type d'outils, visant à évaluer le nombre d'erreurs subsistant dans le programme testé, s'appuie soit sur l'analyse de la structure du programme (complexité), soit sur les techniques de mutation.

6.8. Les comparateurs de fichiers

Ils permettent d'établir les différences qui existent entre deux fichiers. Il suffit de préparer un fichier contenant les sorties souhaitées pour un test donné et de

comparer ce fichier avec les résultats effectivement obtenus à l'issu du test. Si les deux fichiers sont identiques le test sera considéré comme étant réussi.

6.9. Les vérificateurs de programmes

Un vérificateur de programme accepte en entrée un programme, sa spécification et d'autres assertions établies par l'utilisateur. Sa tâche consiste à vérifier que le programme correspond bien à sa spécification. Un tel outil s'appuie sur une spécification formelle de la sémantique du langage de programmation, sur les spécifications formelles et complètes du programme à vérifier et sur des informations supplémentaires données par l'utilisateur.