ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA
MAÎTRISE EN GÉNIE ÉLECTRIQUE
M. ING.

PAR
HAYSSAM ZARAKET

UNE MÉTHODE AUTOMATISÉE POUR LE TEST DES GRANDS LOGICIELS

MONTRÉAL, MARS 2001

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE:

- M. Tony Wong, directeur de mémoire
  Département de génie de la production automatisée à l'École de technologie supérieure

- M. Michel Lavoie, codirecteur
  Département de génie électrique à l'École de technologie supérieure

- M. Pascal Bigras, professeur
  Département de génie de la production automatisée à l'École de technologie supérieure

- M. Abdelkrim Amoura, Ingénieur Logiciel Senior
  Compagnie Ericsson (Département OSS development - PM Product Line)

IL A FAIT L'OBJET D'UNE PRÉSENTATION DEVANT CE JURY ET UN PUBLIC

Le 12 FÉVRIER 2001

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

UNE MÉTHODE AUTOMATISÉE POUR LE TEST DES GRANDS LOGICIELS

Hayssam Zaraket

(Sommaire)

La croissance rapide de l'utilisation des systèmes informatisés dans tous les domaines donne une importance cruciale au problème des anomalies informatiques. De nos jours, les ordinateurs aident les humains dans la plupart des aspects de nos vies quotidiennes, et les applications informatiques sont rendues très puissantes et complexes. Tester une application informatique simple n'est pas une tâche très difficile, mais tester un grand logiciel ayant des millions de lignes de code est une tâche très complexe. Un grand logiciel est complexe et ses interfaces avec les systèmes externes élargissent l'ampleur des activités de test. L'industrie informatique débourse beaucoup d'argent et de temps pour tester de tels logiciels. Elle cherche toujours de nouvelles méthodes pour épargner temps et argent nécessaires à ces tests.

Dans ce mémoire nous allons aborder les difficultés rencontrées lors de test de grands logiciels et les limitations pratiques rencontrées. Nous allons aussi expliquer la nature des anomalies que nous cherchons à débusquer dans le logiciel sous test. Nous montrons comment bâtir un dictionnaire contextuel qui va nous aider dans notre méthode de test afin de trouver les anomalies.

L'objectif principal de ce travail est de présenter une méthode de test pour aider les organisations qui exécutent des tests pour des grands logiciels en tant que tierce partie, c'est-à-dire, une organisation qui n'est pas responsable de la conception de logiciel. Ce qui rend le processus potentiellement complexe impliquant un grand nombre de personnes géographiquement distribuées avec différentes perspectives et compétences. Le processus complet de notre méthode est présenté sous forme détaillée dans ce mémoire.

Ce mémoire présente une méthode et un outil permettant de mieux gérer le processus de test de grands logiciels complexes et de créer des cas de test plus efficaces et reproductibles. Plus important encore, cette méthode va permettre de réduire le temps nécessaire pour élaborer les cas, le plan et la procédure de test. De plus, et contrairement à la méthode de boîte noire, avec cette méthode nous serons en mesure de trouver la raison de l'anomalie trouvée. Toutes les étapes, les difficultés qui peuvent surgir au cours du processus et les livrables à chaque étape seront présentés et détaillés dans ce mémoire.

Une comparaison avec les deux méthodes de test les plus répandues, méthode de boîte noire et méthode de boîte blanche, sera présentée à la fin de ce mémoire pour montrer les avantages de notre méthode. Le type d'anomalies à débusquer dans le cadre de ce mémoire est celui des anomalies informatiques relatives aux problèmes des finitudes de grandeur. Nous recommandons comme travail futur d'appliquer notre méthode pour le problème des « bombes logiques ». Car le test dans des conditions normales ne révèlera pas une telle bombe. Par contre, si les conditions spéciales requises par la « bombe logique » sont rencontrées, le programme agira d'une façon différente. Un bon dictionnaire relatif au contexte de bombes logiques et l'application de notre méthode va aider à trouver les anomalies recherchées. Un autre travail serait d'appliquer notre méthode pour tester les applications informatiques pour le problème de conversion vers l'Euro, un problème que tous les établissements informatiques cherchent à résoudre pour le début de l'an 2001.

AN AUTOMATED METHOD FOR TESTING LARGE SOFTWARE

Hayssam Zaraket

(Abstract)

The rapid growth in the use of computerized systems in all human activities gives a crucial importance to the problem of program bugs. Nowadays, computers are important tools for all aspects of our everyday lives, and computer applications are becoming very powerful and complex. Testing a simple computer application is not a very difficult task, but testing large software with over million lines of code is a very complex task. A large software is complex and its interfaces with external systems widen the scope of test activities. Computer establishments spend much money and time to test such software and are always seeking new methods to save time and money in their tests.

In this thesis we will discuss the difficulties and practical limitations encountered during the test of large software. We will also explain the nature of the bugs we are searching for in the software under test, and will show how to build a contextual dictionary which in our method will help to find the bugs we are looking for.

The principal objective of this work is to provide a method of test to help organisations which, acting as a third party, carry out tests for large software applications. Thus, the process can be complex involving a large number of people geographically distributed with various prospects and competencies. The complete and detailed process of the method will be presented in detail in this thesis.

Also, this work provides a method and a tool allowing better management of the process of testing large and complex software, to create more effective and reproducible test cases and especially to reduce the required time for test cases, plans and procedures creation. Moreover, and contrary to the black box method, this method will allow to find the reason behind the bug found. All the stages, the difficulties that can emerge during the process and the deliverables at each phase will be presented in this thesis.

To show the advantages of our method, a comparison of our method with the two well-known methods of test namely, the black box method and the white box method will be shown at the end of this thesis. We recommend as future work to apply our method to the problem of the so-called « logic bombs » because testing under normal conditions does not reveal such a bomb, but should the special conditions required by that bomb occur, the program performs differently from what is expected. A good dictionary related to the context of logic bombs and application of our method will help find the bugs sought. Another work area would be to apply our method to test computer applications for the problem of conversion towards the Euro currency. A problem that the computer industry seek to solve for the beginning of year 2001.

REMERCIEMENTS

Je tiens à remercier toutes les personnes qui, de près ou de loin, m'ont soutenu dans la réalisation de ce travail. Particulièrement, je voudrais exprimer ma profonde gratitude envers Dr Tony Wong, professeur au département du génie de la production automatisée à l'École de technologie supérieure à Montréal, qui a dirigé ce travail. Je remercie également M. Michel Lavoie qui était mon professeur-cotuteur. J'apprécie sincèrement, non seulement leurs compétences scientifiques, mais aussi la patience, la compréhension et l'aide efficace qu'ils ont témoigné à mon égard.

Je remercie Dr Pascal Bigras, professeur au département de génie de la production automatisée, pour avoir évalué mon travail et pour son engagement en tant que président du jury d'évaluation de ce travail.

Je remercie Dr Abdelkrim Amoura, Ingénieur Logiciel Senior à la compagnie Ericsson (Département OSS development - PM Product Line), pour avoir évalué mon travail d'un point de vue industriel et pour avoir participé comme membre du jury d'évaluation.

Un grand merci à mes parents pour leur soutien moral et leurs encouragements. Je pense en particulier à mon père et ma mère.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

CHAPITRE 1

INTRODUCTION

Il y a trente ans, les pannes d'ordinateur auraient été moins sérieuses en terme de conséquences. Mais de nos jours, les humains utilisent les ordinateurs dans tous les aspects de la vie quotidienne. Cette dépendance croissante sur ces systèmes donne une importance cruciale au problème de la fiabilité des logiciels. En effet, une défaillance de logiciels dans un système de contrôle de centrales nucléaires, lors de manœuvres aérospatiales, sur le contrôle des armes les plus puissantes et au cours de transactions bancaires peuvent engendrer de très grandes pertes économiques, matérielles et humaines.

1.1 Problèmes

La meilleure façon pour garantir la fiabilité des logiciels est de les valider par des tests. Le test de logiciel devient de plus en plus important avec le temps. Ce qui veut dire que les tests doivent être mieux planifier, qu'ils seront réalisés par des professionnels et des spécialistes qui ont une meilleure maîtrise et une bonne connaissance des outils disponibles. Mais malgré cela, on voit toujours des problèmes et des catastrophes reliées à des anomalies logicielles, en voici quelques exemples :

En 1991 [1], DSC Communications of Plano, au Texas, a fait des changements sur trois lignes de code d'un système qui en compte plus de deux millions. DSC a exécuté des tests locaux et régionaux qui ont montré qu'il n'y a pas de problème, ils ont alors décidé de ne pas faire un test de régression. Le changement incluait une erreur qui a

causé des pannes téléphoniques majeurs à Washington, Baltimore, Pittsburg, San Francisco, Los Angeles, et d'autres endroits. La compagnie a eu des problèmes financiers sévères à la suite de cette omission.

Le 4 juin 1996 [2], pendant le vol du lanceur européen Ariane 5, un problème informatique a causé l'explosion et la perte totale de la mission 40 secondes après le décollage. Les rapports ont indiqué que l'affaire a causé une perte d'un demi-milliard de dollars. Ce désastre a été causé par le débordement arithmétique du sous-système de navigation.

Le 08 janvier 1997 [3], le *Dominion*, journal neo-zelandais, annonçait que la production d'aluminium avait été brutalement interrompue à minuit, le 31/12/1996. Le responsable en fut l'un des ordinateurs de contrôle qui "ignorait" que l'année 1996 était bissextile. Deux heures plus tard, à cause du décalage horaire, un incident identique eut lieu en Tasmanie. Dans les deux cas, le redémarrage des installations a coûté plus d'un million de dollars.

Le 3 mai 1998 [3], le London Times a révélé que dans un hôpital Nord londonien, une opération avait dû être reportée parce que le système informatique avait annoncé une rupture de stock concernant les compresses. En fait, il y en avait en nombre suffisant mais leur date limite d'utilisation était située en 2001, ce que l'ordinateur, n'utilisant que les deux derniers chiffres des années, avait interprété comme étant 1901.

Le 5 mai 1998 [3], l'hebdomadaire « 01 Informatique » annonçait que le passage de l'indice Dow Jones au-delà des 9999 points pourrait causer de graves problèmes informatiques à Wall Street (et ailleurs), les ordinateurs correspondants n'ayant pas été programmés, apparemment, pour gérer un indice a 5 chiffres! Bien que ce phénomène n'est pas relié aux problèmes de date, il illustre parfaitement ce qui a été dit au sujet des

capacités limitées des ordinateurs, intrinsèque et/ou causé par les programmeurs.

L'agence *Associated Press* annonçait le 03 janvier 1999 [3] que le système de tarification de 300 taxis de Singapour avait provoqué de mauvaises facturations le 01 janvier 1999 à partir de midi. Un problème similaire aurait été rencontré en Suède.

Comme on a pu le constater, la plupart des anomalies logicielles auraient pu être décelées en réalisant des tests judicieux épargnant ainsi beaucoup de pertes humaines et matérielles. La plupart des compagnies procèdent à des tests avant la sortie de leurs logiciels. Mais ont-ils fait des tests complets en utilisant des bonnes méthodes et des bons outils? Souvent, les compagnies n'ont pas le temps ni le budget nécessaires pour faire ces tests complets, amenant ainsi des situations comme celles que nous venons de citer.

Malgré tous les avancements dans la vérification et la validation automatique, la revue humaine du logiciel est la seule méthode importante pour assurer la qualité des logiciels et surtout détecter les anomalies. Les méthodes automatisées coûtent moins chères et demandent moins de temps d'utilisation. Par contre elles ne sont pas aussi efficaces que la revue du code. Mais le problème qui se pose est qu'il n'est pas possible pour la plupart des compagnies de faire une revue extensive du code à cause du coût et du temps nécessaires. Ce qui est encore plus évident pour les grands logiciels (ceux qui ont plus de 2.5 millions de lignes de code).

Un autre problème qui vient s'ajouter à tous les autres que nous venons de citer, c'est la tendance de nos jours à confier la tâche de validation à un autre établissement indépendant de celui qui a conçu le logiciel. Beaucoup de compagnies préfèrent laisser à une organisation indépendante le soin d'effectuer la validation de leur logiciel. L'idée est qu'une organisation indépendante peut trouver plus d'anomalies que les concepteurs eux-mêmes.

Le test des grands logiciels renferme beaucoup plus de difficultés que le test des petits et moyens logiciels à cause de la complexité des fonctions, des bases de données inter–reliées et des communications avec le matériel. Si on vient rajouter à cela les erreurs et problèmes mentionnés précédemment, le processus de test devient encore plus difficile impliquant beaucoup plus de complications.

Et si comme tout ceci n'était pas suffisant, un autre facteur vient compliquer davantage le processus de test. C'est le facteur de travail collaboratif impliquant un ensemble de personnes géographiquement distribuées, de perspectives différentes et de compétences diverses. Ce facteur augmentera la complexité des activités de test à cause des différences technologiques (normes, expertise), culturelles, psychologiques et linguistiques.

1.2 Proposition

Dans ce travail, nous allons introduire une méthode qui nous permettra de faire la revue du code pour de très grands logiciels tout en gardant un coût raisonnablement bas. Le temps nécessaire à la réalisation des tests est également réduit par la méthodologie proposée. La méthode décrite dans ce travail consiste en un mélange de tests statiques et de tests dynamiques intégrés dans un même processus. De plus, elle utilise les outils appropriés afin de réduire le coût et le temps nécessaire.

L'idée principale de notre méthode consiste à automatiser la revue du code afin de détecter les parties du code qui sont en relation avec les anomalies recherchées. La revue du code est effectuée en créant un dictionnaire contextuel selon la nature de l'anomalie à découvrir et de l'intégrer dans un processus automatisé.

La méthode que nous allons décrire dans ce mémoire est le résultat de plusieurs années d'expérience dans l'industrie. Le logiciel sous test est classé comme un logiciel complexe de grande taille. Notre méthode sera en mesure de résoudre les différents problèmes reliés à l'hétérogénéité de l'équipe de test en fournissant un processus complet pour la communication et la préparation des procédures de test.

1.3 Structure du document

Dans un premier temps, nous introduisons les notions de base reliées aux tests de logiciels et aux différentes méthodes et techniques utilisées dans ce domaine. Ensuite nous expliquons en détails le test qui concerne notre travail, soit le test fonctionnel. Puis, nous dressons et analysons les différents problèmes rencontrés dans le test de grands logiciels effectué par des organisations indépendantes. L'accent est mis sur l'organisation du travail collaboratif inter-organisationnel. Enfin, nous proposons une méthode pour résoudre les difficultés reliées au test de grands logiciels et nous donnons une solution pour les organisations qui exécutent ces tests à titre d'organisations indépendantes.

Ce document est organisé comme suit :

- Le chapitre 2 décrit en détail le test logiciel, montre son importance et présente différentes méthodes de test;
- Le chapitre 3 spécifie le test fonctionnel (boîte noire) et ses différentes techniques;
- Le chapitre 4 présente et analyse les difficultés rencontrées lors des tests de grands logiciels et donne la procédure de notre méthode qui solutionnera les difficultés énumérées;
- Le chapitre 5 spécifie le type d'anomalies recherchées et explique leur origine. En plus, ce chapitre montre et explique en détail l'étape importante de notre méthode de

test, qui consiste à créer un dictionnaire contextuel en fonction de l'anomalie à débusquer.

- Le chapitre 6 présente en détail le processus complet de notre méthode de test et montre ses forces et avantages en la comparant aux deux méthodes fondamentales : méthode de boîte noire et méthode de boîte blanche.
- La conclusion de notre travail présente quelques perspectives reliées à ce travail.

CHAPITRE 2

TEST LOGICIEL

Dans le chapitre précédent, nous avons montré l'importance du test logiciel et présenté un aperçu des objectifs de notre mémoire. Dans ce chapitre, nous allons poursuivre notre discussion plus en détails sur le test logiciel et son importance. De plus, une partie de ce chapitre sera consacrée à la conception des cas de test et à expliquer le modèle séquentiel de cycle de vie du logiciel ainsi que les niveaux de test. Enfin, la dernière partie présente les méthodes de test fondamentales utilisées dans le domaine du génie logiciel

2.1 Définition du test

Le test logiciel est une activité de vérification. Son but est d'établir que l'implantation vérifie les propriétés exigées par la spécification ou de détecter des différences entre elles.

Glenford Myers [4] définit le test comme étant une tâche extrêmement créative et intellectuellement motivante. La créativité requise pour tester un grand logiciel excède la créativité requise pour concevoir ce même logiciel.

Selon la définition donnée par la norme IEEE729 [5], le test est un processus manuel ou automatique qui vise à établir qu'un système vérifie les propriétés exigées par sa spécification ou à détecter des différences entre les résultats produits par le système et ceux qui sont attendus par les spécifications.

Le but fondamental du test de logiciel est de trouver autant d'anomalies que possible dans le code du logiciel. En tant que concepteur de logiciel, il est facile de s'assurer que le logiciel fonctionne. Mais que se produit-il quand l'utilisateur lance une fonction par l'intermédiaire d'une séquence inhabituelle ?

Le test logiciel est un processus organisé qui permet l'identification des différences entre l'état réel et les résultats qu'on attend du système (ANSI/IEEE 1059-1993) [6]. Un test efficace est celui qui permet de détecter les erreurs avec le minimum de temps et d'efforts. Alors le test pourrait être considéré comme un élément d'assurance qu'il faut utiliser intelligemment.

Boris Beizer [7] définit la conception de test comme étant le plus efficace des mécanismes connus de prévention d'erreurs. La réflexion nécessaire pour créer des cas de test utiles peut aider à découvrir et à éliminer des problèmes à chaque étape de développement. Hetzel [8] nous donne une autre définition pour le test logiciel. C'est l'exécution du logiciel ou du système avec l'intention de trouver des erreurs.

Le test de logiciel ne devrait pas être confondu avec le terme déverminage ou «débogage». Le débogage est le processus d'analyser et de localiser les bogues quand le logiciel n'agit pas tel que prévu. Malgré que l'identification de quelques bogues puisse être évidente en manipulant le logiciel, une approche méthodique de test logiciel est un moyen beaucoup plus poussé et efficace. Le débogage est alors une activité qui supporte le test, mais ne peut pas le remplacer.

Pour trouver la bonne approche et la bonne attitude nécessaires pour réaliser des tests, il est important de comprendre les concepts de base : La définition d'une anomalie, le vocabulaire de test, le but d'un testeur, le but des tests et les différents types de test (test d'unité, test d'intégrité, test de système fonctionnel, test d'acceptation, test de

configuration, test de stabilité, test de performance, test de stress, test d'intégrité et de bases de données et test de régression). Dans ce mémoire, nous nous intéresserons au test fonctionnel, c'est-à-dire à la vérification du comportement du logiciel. Le test se distingue des autres activités de vérification telles que la revue de code ou la preuve mathématique en ce qu'il est une activité dynamique. Le test exige que le programme soit exécuté pour que ses résultats puissent être collectés et comparés à la spécification.

2.2 Planification et préparation des tests

Semblable à la plupart des activités complexes, le test de logiciels doit être planifié. Le plan de test décrit le but, l'approche, les ressources et l'échéancier des activités de test (voir chapitre 6). En plus, il identifie les articles à tester, les fonctions à tester, les tâches de test à exécuter, le personnel responsable pour chaque tâche et les risques associés au plan de test.

Le principe de base de test est la sélection des cas de test qui satisfont les critères établis. Mais en plus de ces critères, comme l'indique Paul Rook [9], il peut exister des critères plus généraux auxquels la conformité est nécessaire. Ce sont les critères tels les standards et les pratiques industrielles. Les défaillances reliées à ces critères durant les tests doivent être prises en considération et mentionnées pendant la préparation de la planification des tests.

Il est à noter que les critères de test et les plans de test sont également sujets à une validation. Ceci peut être réalisé par une révision formelle des documents. La planification et la préparation des tests doivent être réalisées par des personnes qui ne sont pas responsables de la conception du logiciel à tester. Les concepteurs sont naturellement concernés par l'exactitude du produit logiciel. Le testeur doit être disposé à trouver des défaillances chez ce même produit logiciel.

2.3 Conception de cas de test

Pour chaque élément à tester et aux différents niveaux de test, un nombre de cas de test seront identifiés. Chaque cas de test spécifie le test d'un critère particulier ou d'une décision de conception. De plus, il spécifie le critère pour le succès du test.

Le standard 829 du ANSI/IEEE [10] définit une spécification de cas de test comme un document ayant les 7 parties suivantes :

- Partie 1 : Identification des spécifications du cas de test (un nom unique qui distingue ce cas de test parmi tous les autres)
- Partie 2 : Éléments de test (une liste d'actions ou de fonctions que ce cas de test va exercer)
- Partie 3 : Spécifications d'entrées (les entrées réelles)
- Partie 4 : Spécifications de sortie (une liste de sorties qui seront les résultats de l'exécution de ce cas de test)
- Partie 5 : Besoins environnementaux (matériel spécial ou logiciel nécessaire pour exercer ce cas de test)
- Partie 6 : Réquisition spéciale d'une procédure (Contraintes sur n'importe quelle procédure qui exerce ce cas de test)
- Partie 7 : Dépendance inter-cas (une liste de cas de test qui doivent être exercés avant que ce cas de test soit exercé)

Tel que décrit par Paul C. Jargenson dans [11], un ensemble de cas de test pour l'élément à tester constitue l'essence du test logiciel. Avant de commencer, on a besoin de clarifier les informations à inclure dans un cas de test. Les informations les plus évidentes sont les données d'entrée. Les entrées sont de deux types : Pre-conditions (Les conditions à priori avant l'exécution de test) et les entrées réelles qui étaient identifiées (lors de la phase de création de cas de test) par une ou des méthodes de test. La deuxième partie d'un cas de test est la sortie prévue. Encore une fois il y a deux types : Post-

conditions et sorties réelles. Supposons par exemple le test d'un logiciel qui détermine le trajet optimal d'un avion, en fonction des contraintes et des données climatiques. Comment savoir si le trajet est optimal ? Une technique utilisée par l'industrie pour répondre à cette question est le test de référence. Le système est testé en présence des usagers experts. Ces experts portent jugements sur les sorties obtenues d'un cas de test exécuté.

Identification du cas de test :			
But :			
Pre-conditions :			
Entrées			
Procédure :			
Post-Conditions :			
Sorties prévues :			
Sortie :			
Dépendances :			
Traçage :			
Date	Version	Résultat	Exécuté par

Figure 2.1 Information typique d'un cas de test

La procédure à suivre pour exécuter le test, les dépendances par rapport à d'autres cas de test et le retraçage des cas par rapport aux spécifications sont des informations importantes dans un cas de test. Le reste des informations (voir figure 2.1) sert à faciliter l'administration du test. Les cas de test ont une identification et une raison d'être. Il est aussi utile d'enregistrer l'historique de l'exécution d'un cas de test c'est-à-dire, le moment, la personne responsable, le résultat (succès/échec) de chaque exécution et la version (du logiciel) sur laquelle le test a été effectué. Nous pouvons conclure que les cas de test sont aussi importants que le code source lui-même. Les cas de test ont besoin d'être développés, révisés, utilisés, administrés et enregistrés.

2.4 Modèle séquentiel du cycle de vie de logiciel

Le développement d'un logiciel passe par plusieurs phases, dans lesquelles des activités comme la conception et le test sont fréquents. Un des modèles pour décrire le cycle de développement de logiciel est le modèle V (figure 2.2).

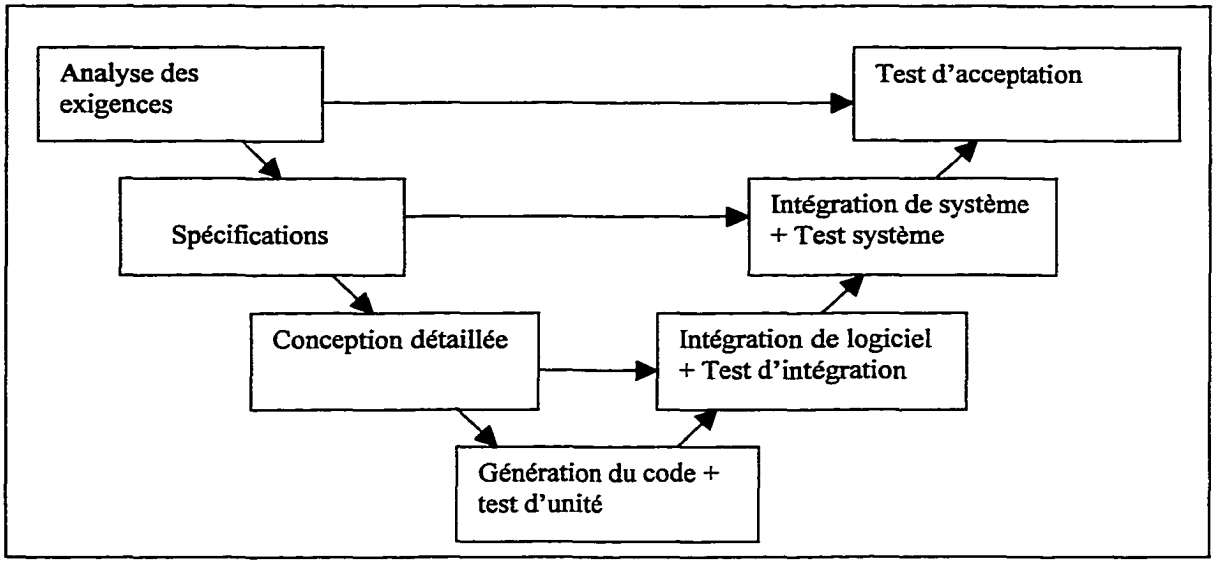


Figure 2.2 Modèle V de cycle de vie de développement de logiciel [9]

Dans ce modèle, les axes horizontaux (de gauche à droite) indiquent le passage du temps. Les axes verticaux descendants à gauche montrent l'élaboration et la décomposition du logiciel proposé durant la conception et les axes verticaux ascendants à droite montrent la composition du logiciel durant l'intégration. Dans le modèle V, chaque phase de développement de logiciel (à gauche) correspond à une phase de test (à droite). Ci-dessous est une description de chaque phase dans le modèle V :

- La phase d'analyse des exigences où les exigences du logiciel sont recueillies et analysées pour produire des spécifications complètes et non ambiguës de ce que le logiciel doit faire ;

- La phase des spécifications où l'architecture de logiciel pour la mise en place des exigences est conçue et spécifiée identifiant les composants dans le logiciel et les relations entre les composants ;
- La phase de conception détaillée où la mise en place détaillée de chaque composant est indiquée ;
- La phase de génération de code et de test d'unité où chaque composant du logiciel est codé et testé pour vérifier qu'il met en application correctement la conception détaillée ;
- La phase d'intégration de logiciel où des groupes de composants de logiciel sont intégrés et testés jusqu'à ce que le logiciel fonctionne dans son ensemble ;
- La phase d'intégration de système où le logiciel est intégré au produit global et testé ;
- La phase de test d'acceptation où des tests sont appliqués et utilisés pour valider le fonctionnement correct du logiciel selon les exigences définies.

Les caractéristiques de logiciel seront produites à partir des trois premières phases de ce modèle de cycle de vie. Les quatre phases restantes impliquent le test du logiciel à des niveaux différents.

2.5 Niveaux de test

Les divers niveaux de test forment une hiérarchie illustrée à la figure 2.3. Le standard ANSI/IEEE 1059-1993 [6] décrit ces niveaux de test comme étant le test d'unité, d'intégration, de système et d'acceptation.

Le test d'unité, aussi appelé test de modules, est au niveau le plus bas dans la hiérarchie. L'objectif général de ce niveau de test est de trouver les bogues dans la logique, dans les données et dans les algorithmes de chaque module et ce, individuellement [10].

Boris Beizer [7] donne une grande importance au test d'unité et considère un test d'unité efficace comme étant la fondation des phases subséquentes des tests. Aucun effort de test d'intégration ou de système ne peut compenser un test d'unité inadéquat. Selon le standard ANSI/IEEE STD 1059-1993, le test d'unité est défini comme un "test effectué pour vérifier la mise en place de la conception d'un élément de logiciel (par exemple, unité ou module)".

Après le test d'unité, viennent les tests d'intégration. Ces derniers sont effectués pour trouver les bogues d'interfaces entre les modules.

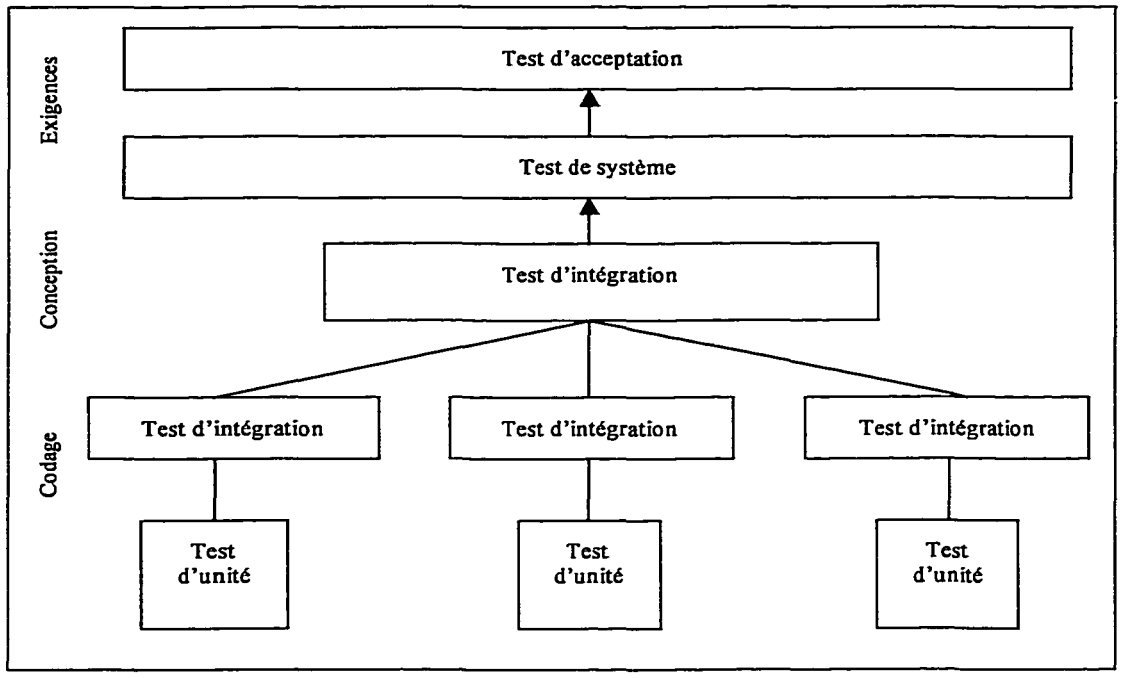


Figure 2.3 Niveaux de test [35]

En remontant dans la hiérarchie, le niveau suivant les tests d'intégrations est le test d'intégration puis de système. Ce dernier est le processus de tester un système formé de matériels et de logiciels intégrés afin de vérifier le système par rapport aux exigences définies [6].

Une partie du test de système est le test de régression (chapitre 3). Le test de régression est le type de test le plus complexe utilisé pour déterminer la satisfaction des exigences face à des modifications et à des changements. Le test de régression est fréquemment utilisé durant le développement de logiciels où de nouvelles versions du produit sont en phase de réalisation.

Finalement, au plus haut niveau de la hiérarchie vient le test d'acceptation. Le but de ce test est de déterminer si le logiciel répond aux spécifications du client. Les tests

Alpha et Bêta sont des variantes du test d'acceptation. Le test Alpha se rapporte généralement aux tests faits par les concepteurs de logiciel eux-mêmes. Le test bêta se rapporte généralement aux tests faits après l'accomplissement du test alpha. Le test bêta est réalisé par des personnes autres que les concepteurs. C'est-à-dire, des testeurs officiellement indiqués ou encore des clients qui font partie d'un programme de test Bêta.

2.6 Méthodes de test

Tout comme les niveaux de test, il existe différentes méthodes de test. Il existe deux méthodes fondamentales en génie logiciel : méthode de test fonctionnel connue sous le nom boîte noire (chapitre 3) et méthode de test structurel connu sous le nom boîte blanche. Ces deux méthodes nous donnent des techniques pour identifier les cas de test. Mais avant de procéder à la description de ces deux méthodes fondamentales, nous allons d'abord présenter la classification des méthodes de test. Généralement, on distingue deux classes de méthodes de test : Les méthodes de test statiques et les méthodes de test dynamiques.

2.6.1 Méthodes de test statiques

Les méthodes de test statiques consistent en l'analyse textuelle du code source afin d'y détecter les erreurs. Il s'agit par exemple d'effectuer une simple revue du code. En d'autres termes et pour faire un lien avec la définition de test donnée par la norme IEEE729 [5], dans un test statique on analyse le code du logiciel afin de s'assurer qu'il vérifie certaines propriétés exigées par les spécifications de l'application.

2.6.2 Méthodes de test dynamiques

Les méthodes de test dynamiques consistent en l'exécution du programme à vérifier à l'aide d'un certain nombre de scénarios d'utilisation de ce programme. Elles visent à détecter des erreurs en confrontant les résultats obtenus par l'exécution du programme à ceux attendus par les spécifications de l'application. Il existe plusieurs méthodes de tests dynamiques :

1. Méthode structurelle ou «boîte blanche »
2. Méthode fonctionnelle ou «boîte noire »

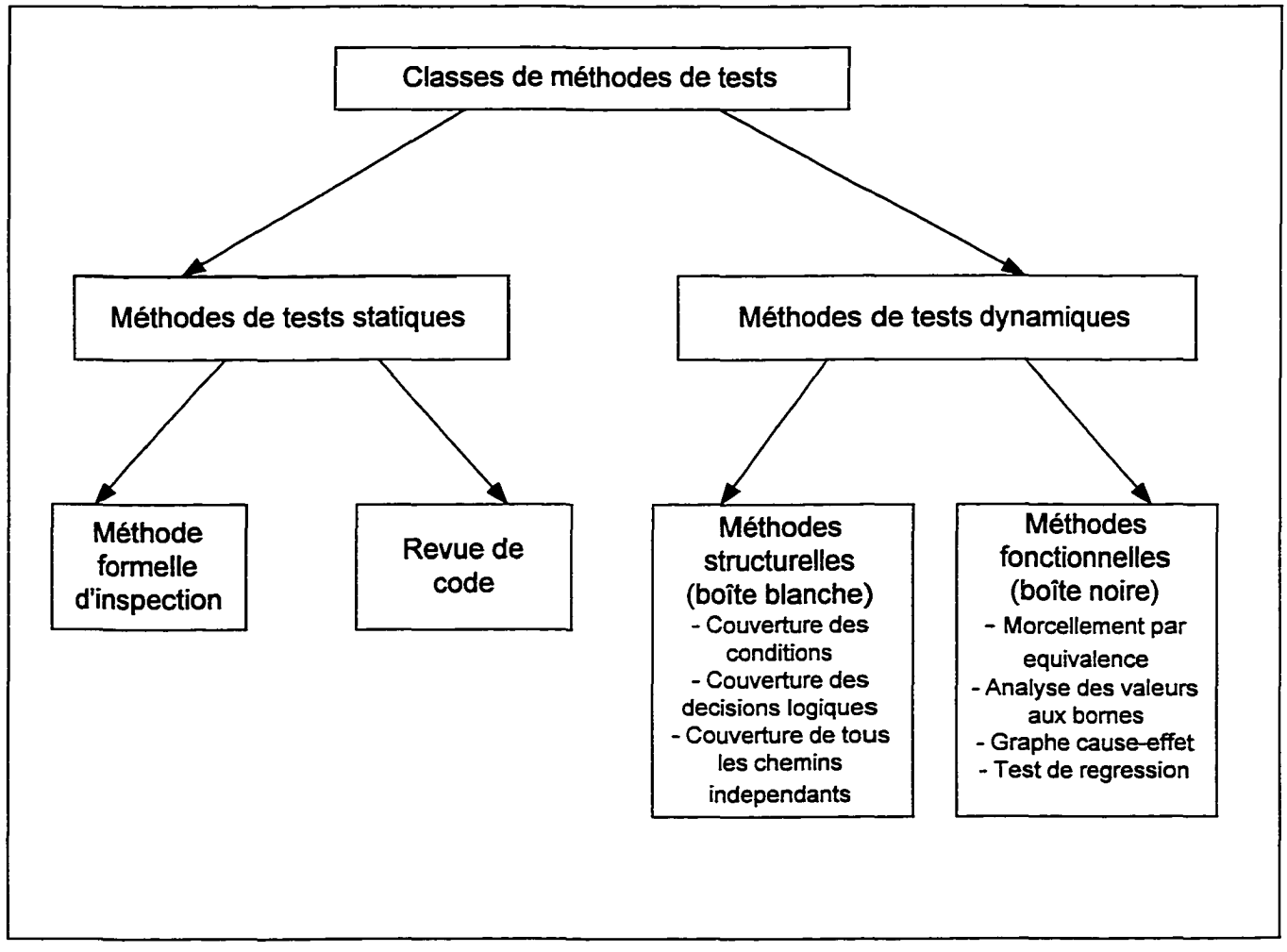


Figure 2.4 Différentes méthodes de test

2.6.3 Méthode de test fonctionnel

Il s'agit de la méthode la plus utilisée pour identifier les cas de test. L'idée de base est très simple : une certaine fonction est attendue d'un système. Le cas de test vérifie que le système exécute cette fonction attendue. Cette notion est la plus utilisée en ingénierie où le système est considéré comme une boîte noire. Ceci nous amène au terme Test Boîte Noire, dans lequel le contenu de la boîte noire est inconnu et la fonction de la boîte noire est comprise complètement en terme de ses entrées et ses sorties.

Selon l'IEEE [12], un test fonctionnel est celui qui ignore le mécanisme interne d'un système et regarde seulement les sorties produites en réponse aux entrées choisies et aux conditions d'exécution.

Les techniques utilisées sont semblables à la physique expérimentale des particules. Les physiciens dirigent les faisceaux de particules d'énergie élevées (habituellement des électrons ou des positrons) sur des cibles faites du matériel sous étude. Les physiciens examinent les débris des collisions de particules ("ce qui sort de l'autre côté "), pour former des conclusions sur la construction interne du matériel. De même, les testeurs utilisant la méthode boîte noire bombardent le système sous investigation avec des cas de test et examinent les résultats obtenus.

Ce type de test a pour objectif de trouver les erreurs dans les catégories suivantes :

- Les fonctions incorrectes ou manquantes ;
- Le respect des exigences ;
- Les interfaces manquantes ;
- Les erreurs dans des structures de données ou l'accès aux bases de données externes ;
- Les problèmes de performance de l'application ;
- Les conditions initiales ou finales d'une fonction.

L'intérêt principal des tests de boîte noire est qu'ils valident les fonctionnalités du logiciel plus que la qualité du développement. Ils sont donc plus proches des préoccupation quotidiennes des utilisateurs.

2.6.4 Méthode de test structurel

Le test structurel est une autre méthode fondamentale pour identifier des cas de test. Afin de créer un contraste entre le test fonctionnel et le test structurel, ce dernier est appelé souvent test boîte blanche. Selon l'IEEE [13] le test boîte blanche tient compte du mécanisme interne d'un système ou d'un composant.

Les tests en boîte blanche sont basés sur un examen du texte du programme : les tests sont sélectionnés de manière à remplir certaines exigences de couverture de l'implantation (toutes les instructions, tous les chemins exécutables, etc.). S'ils sont faciles à mettre en œuvre, les tests en boîte blanche souffrent cependant d'un défaut rédhibitoire : cette stratégie ne permet de tester que ce qui figure dans le programme, mais ne permet pas de mettre à jour des oublis du programme par rapport aux spécifications.

Avec le test boîte blanche on peut vérifier si :

- Tous les chemins ont été parcourus au moins une fois ;
- Les conditions vrai/faux ont été validées ;
- Les boucles sont exécutées correctement ;
- Les valeurs limites sont correctement définies ;
- Les structures de données internes sont valides ;
- Il n'y a pas de fuite de mémoire.

Afin d'illustrer ce que nous venons d'expliquer nous fournissons le schéma de la figure 2.5. Cette dernière montre la relation entre les différents niveaux de test et les différentes méthodes de test au cours du cycle de vie de logiciel.

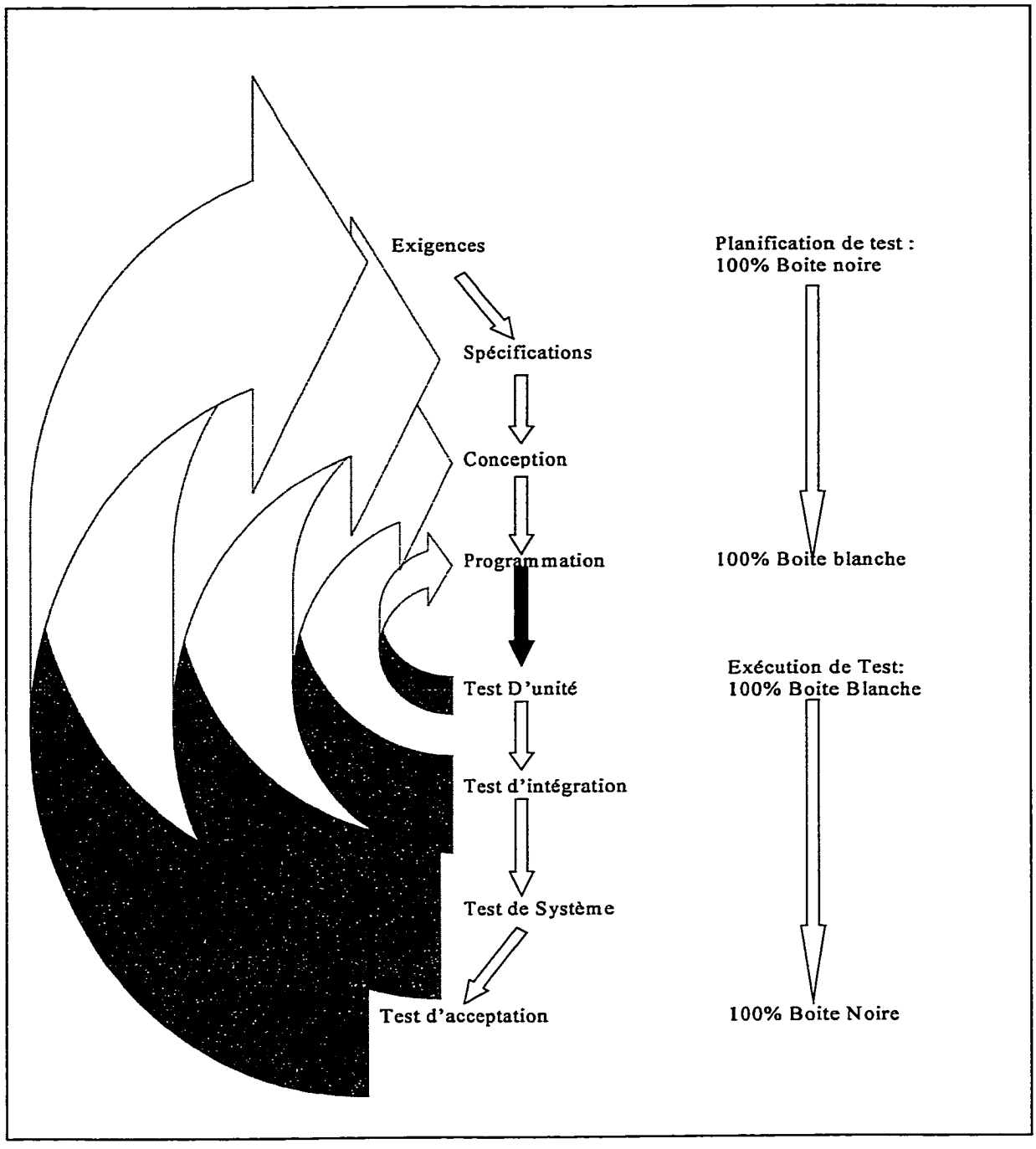


Figure 2.5 Relation entre les niveaux de test et les méthodes de test

2.7 Conclusion

Le but de ce chapitre est de présenter le domaine des tests de logiciel et de montrer les différentes notions et méthodes utilisées. Nous avons défini le test de logiciel et montré son importance en résumant les opinions de la littérature dans le domaine de test de logiciel. Par la suite, nous avons montré comment concevoir un cas de test et dresser ses composants.

Nous avons consacré deux sections de ce chapitre pour préciser le rôle et la place du test de logiciel dans le cycle de développement de logiciel et les différents niveaux de test existants.

La dernière section de ce chapitre a été un survol des méthodes fondamentales présentées dans la littérature et utilisées dans le monde du test logiciel. Dans le chapitre suivant, nous abordons l'une de ces méthodes plus en détail, soit la méthode appliquée dans ce projet, la méthode de test fonctionnel.

CHAPITRE 3

TEST FONCTIONNEL

Les tests effectués dans ce travail sont des tests fonctionnels. Normalement un test fonctionnel est une activité orientée boîte noire. Dans le chapitre précédent nous avons dressé les différents niveaux et méthodes de test. Ce chapitre, expliquera plus en détail le test qui nous concerne particulièrement soit le test fonctionnel. De plus nous étudierons les différentes techniques qui y sont rattachées.

3.1 Définition du test fonctionnel

Le test fonctionnel a une approche un peu différente du test structurel. Ce dernier est plus technique et il est exécuté par des spécialistes qui sont plus proche des détails du système. Le test structurel est exécuté avant le test fonctionnel et sur des sous-systèmes qui pourraient être beaucoup plus petits que le système au complet.

Le testeur de système (test fonctionnel) teste le système assemblé au complet. La quantité d'information et les détails sont tels que le testeur ne peut s'enfoncer dans la complexité de chaque composant. Au lieu de ceci, le testeur de système prend la grande perspective et teste le système au complet comme une seule entité. Ce qui pourrait inclure des entrées de données ou des commandes massives et des combinaisons de réponses massives.

Steve Maquire compare le test fonctionnel comme si on devait déterminer l'état mental de quelqu'un. Il dit : « Tu poses une question ; tu écoutes les réponses et tu porte un jugement. À la fin, tu n'es jamais certain parce que tu ne sais pas qu'est-ce qui se passe dans la tête de l'autre personne » [14]. En d'autre mot, il faut toujours procéder avec la prétention que le logiciel comporte des bogues.

3.2 Test de boîte noire

Le test fonctionnel est nécessairement de type boîte noire. Le test de boîte noire implique la sélection des données de tests et l'interprétation des résultats de test basées sur les propriétés fonctionnelles d'une partie du logiciel. Le test de boîte noire ne devrait pas être réalisé par l'auteur du programme qui connaît son mécanisme de fonctionnement [15]. Dans les nouvelles approches de test, les logiciels sont donnés à une tierce partie pour fin de validation.

Les tests de boîte noire sont des tests fonctionnels (au sens large) conçus pour s'assurer que les exigences du système sont respectées et que les spécifications sont accomplies. Le processus implique la création des cas de test pour les utiliser en évaluant le bon fonctionnement de l'application [16].

Le test de boîte noire ignore le mécanisme interne d'un système ou d'un composant et se concentre seulement sur les sorties produites en réponse aux entrées choisies et aux conditions d'exécution [13]. Dans ce type de test le testeur visualise le logiciel comme une boîte noire et en tant que tel, les fonctionnements internes du logiciel sont inconnus. L'outil principal utilisé dans le test de boîte noire est la spécification du logiciel. C'est-à-dire, le testeur essaye de déterminer quelle entrée donnera une sortie du logiciel différente de ce que les spécifications exigeraient.

Pour réaliser le test boîte noire avec succès, les relations entre les nombreux différents modules dans le système doivent être comprises. Les méthodes de test boîte noire ont pour but de rechercher :

- Le respect des exigences ;
- Les fonctions incorrectes ou manquantes ;
- Les incohérences au niveau de l'interface ;
- Les erreurs dans les structures de données ou dans l'accès aux bases de données externes ;
- Les problèmes de performance de l'application ;
- Les erreurs aux niveaux des conditions initiales ou finales d'une fonction.

Les tests sélectionnés par cette stratégie garantissent une bonne couverture du domaine d'entrée du logiciel et des oublis par rapport aux spécifications. De plus, lors de modifications du logiciel ne remettant pas en cause les spécifications, il est possible de conserver les tests précédemment sélectionnés pour vérifier le logiciel modifié.

3.3 Morcellement par équivalence

La plupart des spécialistes utilisent la notion «d'équivalence » en faisant des tests. Si un cas de test donne un certain comportement que ce soit valide ou non, on suppose que d'autres cas de tests équivalents vont se comporter de la même façon. Alors ils n'ont pas besoin d'être testés. Par exemple, si on essaie d'entrer une chaîne de caractère non numérique spécifique dans un champ d'entrée de données numériques et cette chaîne non numérique est rejetée, on pourrait assumer que n'importe quelle autre chaîne non numérique équivalente va être aussi rejetée.

Le morcellement par équivalence est une méthode de test de la technique boîte noire qui divise le domaine d'entrée d'un programme en des classes de données à partir desquelles des cas de test vont être conçus [17]. Le problème majeur dans le test de

morcellement par équivalence est la détermination d'un sous-ensemble de cas de test qui a la plus grande probabilité de détection d'erreurs.

Une classe d'équivalence est un ensemble de conditions d'entrées ou des valeurs de données d'entrée qui produisent le même résultat à la sortie. Si un seul cas de test dans la classe d'équivalence fonctionne correctement, on pourrait assumer que le reste va avoir le même comportement (i.e. fonctionne correctement). Alors le reste des cas de test n'a pas besoin d'être testé.

Cette méthode divise l'entrée d'un programme en classes de données. La conception de cas de test est basée sur la définition d'une classe équivalente pour une entrée particulière. Une classe d'équivalence représente un ensemble de valeurs d'entrée valides et invalides. Voici les directives pour le morcellement par équivalence :

1) Si une condition d'entrée indique un intervalle, une classe d'équivalence valide et deux classes invalides sont définies.
2) Si une condition d'entrée exige une valeur spécifique, une classe d'équivalence valide et deux classes invalides sont définies.
3) Si une condition d'entrée indique un membre d'un ensemble, une classe d'équivalence valide et une classe invalide sont définies.
4) Si une condition d'entrée est booléenne, une classe valide et une invalide sont définies.

3.4 Analyse de valeurs aux bornes

Comme le terme l'indique, cette méthode se concentre sur la conception de cas de test qui examinent les bornes supérieures et inférieures de la classe d'équivalence. Ces cas de test ne sont pas basés seulement sur des conditions d'entrée comme dans la méthode de morcellement par équivalence mais aussi sur des conditions de sortie [17].

L'analyse de valeurs aux bornes est complémentaire au morcellement par équivalence. Plutôt que de choisir des valeurs d'entrée arbitraires pour diviser la classe d'équivalence, le concepteur de cas de test choisit des valeurs aux bornes de la classe. En outre, l'analyse de valeurs aux bornes encourage également les concepteurs de cas de test à regarder les états de sortie et à concevoir des cas de test pour les conditions aux bornes à la sortie.

Il est important de noter que les programmes qui échouent avec des valeurs différentes des valeurs aux bornes vont échouer habituellement aux bornes aussi [18]. L'analyse de valeurs aux bornes est basée sur le fait que les défauts tendent à se regrouper autour des bornes dans les programmes.

Comme avec d'autres formes de test, nous faisons quelques suppositions dans l'analyse de valeurs aux bornes. Nous supposons que le programme divise l'entrée en un ensemble de domaines dans lesquels le comportement du programme est semblable. Pour cette raison, nous supposons que si un élément d'un certain domaine d'entrée produit une erreur alors une erreur semblable sera produite pour tous les éléments de ce domaine d'entrée. Nous supposons également que si un cas de test ne donne pas d'erreur alors tous les autres éléments dans le domaine ne donneront pas d'erreur.

Selon ces suppositions, nous essayons de couvrir la structure de l'entrée du programme et non pas la structure du programme. Ceci n'exige aucune connaissance de la syntaxe du programme - seulement la connaissance de ses spécifications. Ceci nous permet de concevoir les cas de test dès la disponibilité des spécifications. Par contre, les techniques de boîte blanche exigent que le système soit mis en application avant que nous puissions commencer à définir des cas de test. Par exemple,

1. Pour un intervalle de valeurs bornées par a et b, tester (a-1), a, (a+1), (b-1), b, (b+1).

2. Si les conditions d'entrée indiquent un certain nombre de valeurs n, tester avec (n-1), n et (n+1) valeurs d'entrée.

3. Appliquer 1 et 2 pour les conditions de sortie (par exemple, produire une table de taille minimale et maximale).

4. Si les structures de données internes du programme ont des bornes (par exemple, taille de mémoire tampon, table de limites), utiliser des données d'entrée pour exécuter les structures sur les bornes.

Une abstraction proposée par Boris Beizer [19], utilise le concept de bornes ouvertes et bornes fermées. Une borne fermée est celle dont la borne elle-même (i.e. le point de la borne) est incluse dans l'intervalle des valeurs valides, par conséquent, une borne ouverte est celle où la valeur de la borne n'est pas incluse dans l'intervalle des valeurs valides.

Par exemple, dans la situation où $0 < x <= 1$, la valeur de la borne inférieure «zéro» est une borne ouverte pour x et la valeur de la borne supérieure « un » est une borne fermée.

Si la borne est ouverte, Beizer [19] suggère de tester sur les bornes (qui est une valeur invalide à l'extérieur de l'intervalle), et juste à l'intérieur des bornes. Si la borne est fermée, il suggère de tester sur les bornes (qui est maintenant une valeur valide) et juste à l'extérieur des bornes. Dans le deux cas, deux tests sont suffisants au lieu de trois pour chaque borne. Dans la situation où $0<x<=1$, et le plus grand incrément dans le système de numération est 0.01, les tests pour la borne inférieure sont 0.00 et 0.01. Les tests pour la borne supérieure sont 1.00 et 1.01.

3.5 Graphe Cause-effet

Le graphe cause-effet est une technique de boîte noire. Les graphes de cause-effet modélisent les descriptions narratives complexes du logiciel comme étant des circuits logiques numériques qui peuvent facilement être utilisés pour développer des cas de test fonctionnels [20]. Chaque circuit est une représentation imagée de la sémantique décrite dans les spécifications. L'information sémantique dans les graphiques de cause-effet est traduite en tableaux de décision qui sont utilisés pour construire les cas de test.

Si les spécifications contiennent des combinaisons de conditions d'entrée, alors il serait bon de commencer par la méthode de graphe cause-effet [4]. Le graphe cause-effet [20] est une technique qui aide à choisir d'une façon systématique, un grand ensemble de cas de test. Elle a un effet secondaire bénéfique en précisant l'imperfection et les ambiguïtés dans les spécifications.

Une faiblesse des deux autres méthodes (morcellement par équivalence et analyse de valeurs aux bornes) est qu'elles ne considèrent pas les combinaisons des états d'entrée-sortie potentielles [4]. Les graphes cause-effet relient des classes d'entrée (causes) aux classes de sortie (effets) donnant un graphique très visuel.

Algorithme de la méthode graphes cause-effet :

1) Des causes et des effets sont énumérés pour des modules et une identification est assignée à chacun ;
2) Un graphique de cause-effet est développé (des symboles spéciaux sont exigés) ;
3) Le graphique est converti en table de décision ;
4) Des règles de table de décision sont converties en cas de test.

Il y a quatre configurations fondamentales et deux formes négatives rarement utilisées.

L'**identité** définit une situation dans laquelle le noeud Y est vrai si le noeud X est vrai. En termes booléens, si X = 1, Y = 1, sinon Y = 0.

Le **ET** définit une circonstance où X et Y doivent être vrais pour que Z soit vrai. Encore, dans la logique booléen, Z = 1 seulement si X = 1 et le Y = 1, sinon Z = 0.

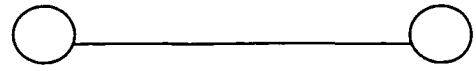
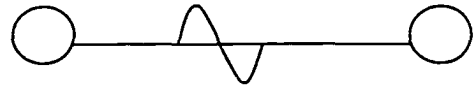
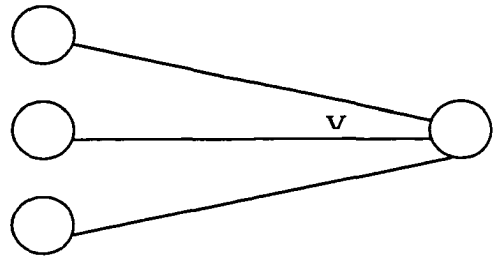
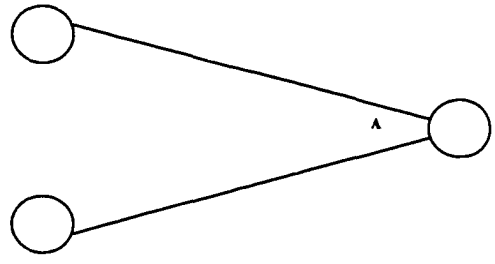
Le **OU** définit une condition dans laquelle Z doit être vrai si X ou Y est vrai. Dans le format booléen, Z = 1 si X = 1 ou Y = 1, sinon Z = 0.

Le **NON** définit l'exemple où Y est vrai seulement si X est faux. Dans la logique booléenne, Y = 1, si X = 0, sinon Y = 1.

Le tableau suivant montre les quatre configurations de base utilisées dans la méthode du graphe cause-effet.

Tableau 3.1

Symboles des quatre configurations de base du graphe cause-effet [36]

IDENTITÉ	
NON	
OU	
ET	

Pour illustrer un petit graphe, prenons un exemple [4]. Supposons qu'on a les spécifications suivantes : Le caractère dans la première colonne doit être un A ou un B. Le caractère dans la deuxième colonne doit être un chiffre. Dans cette situation la mise à jour du fichier est faite. Si le premier caractère est incorrect, un message E1 est généré. Si le deuxième caractère n'est pas un chiffre, un message E2 est généré.

Les causes sont :

1- Caractère dans la colonne 1 est A
2- Caractère dans la colonne 1 est B
3- Caractère dans la colonne 2 est un chiffre

Et les effets sont :

70- Mise à jour faite
71- Message E1 généré
72- Message E2 généré

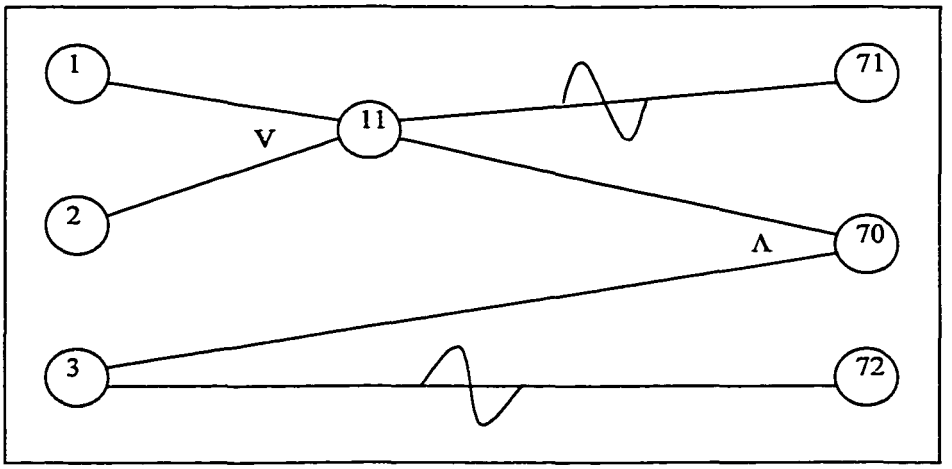


Figure 3.1 Exemple de graphe cause-effet [4]

On remarque que ce graphe (figure 3.1) représente les spécifications en essayant tous les états possibles des causes et en vérifiant que les effets correspondent aux bonnes valeurs. Ensuite on développe le graphe cause-effet et on le convertit en tableau de décision pour générer, par la suite, les cas de test.

3.6 Test de régression

Un test de régression est un re-test compréhensif de tout le système. Un test de régression est fait après l'application des modifications à un système existant. Il a pour but de vérifier le fonctionnement du système dans son entier. Toutes les fonctions testées d'un système sont re-testées, car rien ne garanti qu'elles vont toujours fonctionner de la même façon après les changements introduits.

Le but primaire du test de régression est de montrer que les perfectionnements du système et l'entretien routinier n'affectent pas la fonctionnalité initiale du système. Le but secondaire est de montrer que les changements font ce qu'ils sont destinés faire. Le test de régression est répété chaque fois que des changements sont appliqués au système. Ceci signifie que les cas de test de régression ne devraient pas être jetés après chaque test.

Le test de régression est la forme de test la plus favorable à l'automatisation. Dans cette étape le robot d'assurance qualité logiciel devrait être utilisé pour concevoir et établir les séquences de test automatisé. Les séquences peuvent alors être mises en valeur et réutilisées pour chaque test de régression ultérieur.

La régression joue un rôle important dans les tests de boîte noire. En voici les objectifs [21] :

- Déterminer si la documentation du système demeure à jour ;
- Déterminer si les données de test et les conditions de test du système sont à jour ;

- Déterminer si les fonctions du système testées antérieurement fonctionnent correctement après l'introduction de changements au système.

En pratique, les tests de régression sont appliqués à peu près 1 fois pour chaque 100 fois que la théorie l'exige, à cause des limitations de temps et de ressources. En plus, les tests de régression ne sont pas faits parce que les gens ne voient pas les risques. Selon Steve McConnel [22], «La plupart des gens laissent tomber le test de régression parce qu'ils pensent que leurs changements sont innocents».

En 1991 [1], DSC Communications of Plano, en Texas, a fait des changements sur 3 lignes de code d'un système de deux million de lignes de code. DSC a exécuté des tests locaux et régionaux qui ont montré qu'il n'y a pas de problème et n'ont pas fait de test de régression. Le changement incluait une erreur d'un caractère (un « d » mal codé comme un « 6 ») qui a causé des pannes téléphoniques majeures à Washington, Baltimore, Pittsburg, San Francisco, Los Angeles, et d'autres endroits. DSC a frôlé la faillite à cause du manque d'un test de régression !

Considérer la situation suivante : Un nouveau système est envoyé en test et 100 cas de test sont exécutés. Parmi ces 100 cas, 93 cas de test ont passé et 7 ont échoué. Le testeur génère un rapport de problèmes pour les 7 tests qui n'ont pas passé et les envoie au concepteur.

Après une période de temps, le concepteur produit une nouvelle version du logiciel. Les 7 cas de test qui ont échoués avant sont re-exécutés et 6 passent maintenant. Un autre rapport de problème est soumis pour la dernière erreur récalcitrante. Le concepteur répond avec une troisième version et le cas de test en question passe.

À ce stade ci, la question qui se pose est la suivante, tous les 100 cas de test ont passé, est-ce que le logiciel est maintenant sans erreurs ?

Selon la théorie de régression, le fait qu'un cas de test a passé contre une version antérieure du logiciel ne garanti pas qu'il va marcher pour la version finale. De nouvelles erreurs pourraient avoir été introduites en faisant les corrections des erreurs antérieures.

La figure 3.2 montre que la correction de l'anomalie au module 5 a créé une nouvelle anomalie dans le module 4 car les modules 4 et 5 sont reliés par des parties du code communes. Il y a donc eu régression de l'application, c'est-à-dire que l'application contient des anomalies cachées. Le seul moyen de les détecter est de tester de nouveau l'ensemble de l'application après chaque correction.

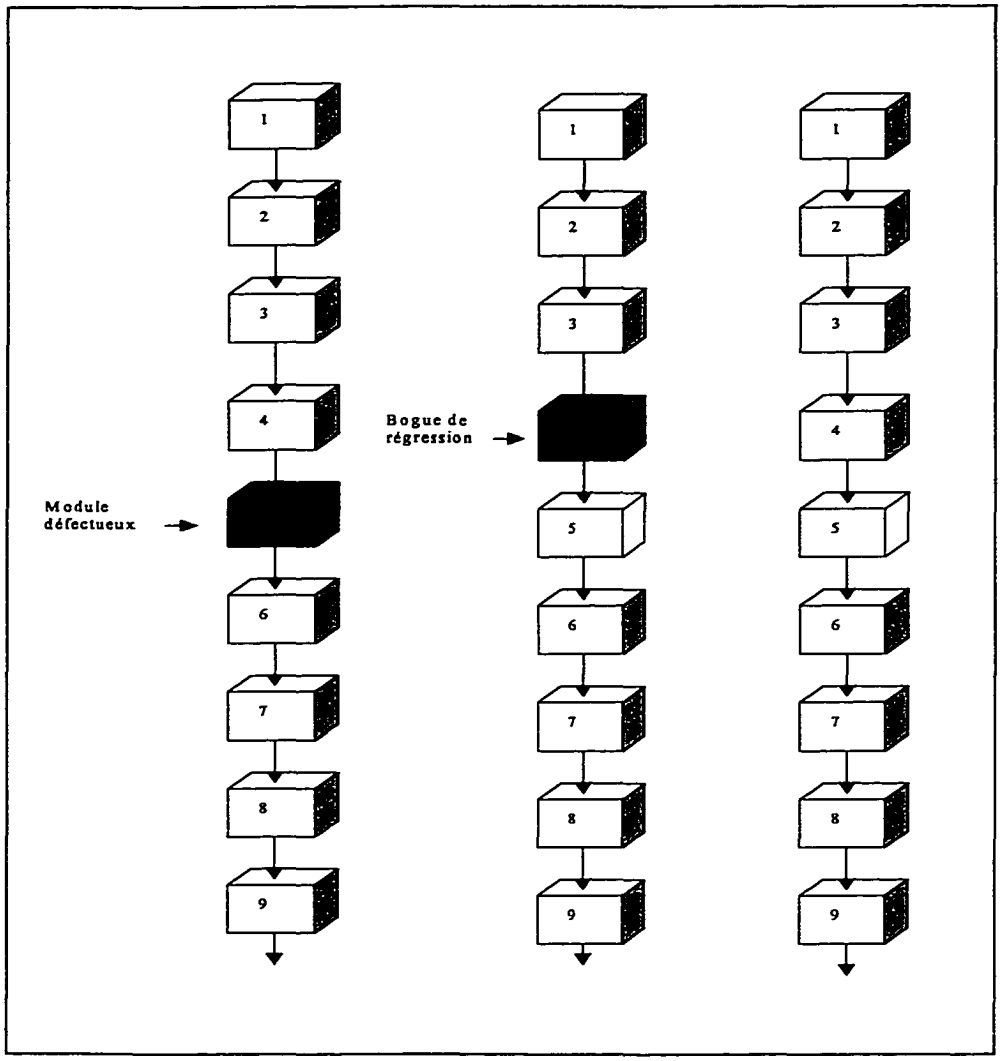


Figure 3.2 Bogue de régression [16]

La régression est l'un des problèmes les plus complexes à résoudre car elle peut apparaître à tout moment et sous des formes diverses.

3.7 Test de performance

Beaucoup de programmeurs ont des objectives de performance et d'efficacité. Tel que le temps de réponse de l'application sous certaines conditions de charge de travail et de configuration [4].

Le but du test de performance est de mesurer la performance sous charge. Un autre nom utilisé pour le test de performance est le test de capacité.

3.8 Conclusion

L'objectif de ce chapitre est d'exposer les différentes techniques du test fonctionnel. Ce dernier est le test appliqué dans ce travail. Les définitions, buts et forces de ces techniques ont été identifiées.

Ce chapitre a montré en détail la nature et des techniques de test reliées aux tests fonctionnels que nous allons appliquer dans ce mémoire. Dans le chapitre suivant nous allons discuter plus profondément de la nature du logiciel sous test et montrer les problèmes rencontrés dans le test d'un tel logiciel et les solutions proposées.

CHAPITRE 4

TEST DE GRANDS LOGICIELS

La méthode de test que l'on présente dans ce mémoire s'adresse particulièrement aux grands logiciels. Le test de ces derniers renferme beaucoup plus de difficultés et de complications que le test d'un petit ou moyen logiciel. Dans ce chapitre nous allons montrer la nature des grands logiciels et les difficultés présentes en les testant. De plus, nous allons discuter de l'impossibilité d'un test complet et l'implication des problèmes de type NP et d'explosion combinatoire. Nous terminons le chapitre par une discussion sur la solution des problèmes de test de grands logiciels tout en donnant un aperçu de notre méthode de test.

Les grands logiciels ont été développés pour résoudre les grands problèmes et peuvent être caractérisés non seulement par la taille mais aussi par le coût, la complexité, et les interfaces, etc. Par exemple, les grands logiciels sont coûteux à développer et maintenir. Un grand logiciel est complexe et ses interfaces avec les systèmes externes élargissent le but des activités de test. Les grands logiciels s'adressent aux grands problèmes importants. Sinon, leurs coûts ne sont pas justifiables. Voici quelques exemples de grands systèmes :

- Un logiciel qui contrôle le trafic ferroviaire ou aérien;
- Un logiciel qui analyse les réacteurs nucléaires et détermine comment ils vont agir durant leur vie utile;
- Un logiciel de navigation utilisé par les navettes spatiales.

4.1 Définitions

Edward Yourdon [23] définit les petits, et les moyens logiciels comme suit :

4.1.1 Petit logiciel

Un petit logiciel offre un service simple, comme un logiciel d'inventaire, un logiciel de comptabilité ou un logiciel de facturation. Certainement, « petit » ne veut pas dire sans importance. Typiquement, un petit logiciel est en bas de 500,000 lignes de code.

4.1.2 Logiciel moyen

Un logiciel moyen est celui qui offre plusieurs services. La différence majeure entre un petit logiciel et un logiciel moyen est la fonctionnalité et l'intégration de la fonctionnalité dans un module orienté usager. Typiquement, un logiciel moyen est de 500,000 à 2 millions de lignes de code.

4.1.3 Grand logiciel

Un grand logiciel offre beaucoup de services. La différence majeure entre un logiciel moyen et un grand logiciel est la complexité de ses fonctions. La complexité peut être causée par des algorithmes de calcul complexe ou par des relations de données complexes. Typiquement, un grand système contient plus de 2.5 millions de lignes de code.

4.2 Logiciel sous test

Le logiciel sous test utilisé dans ce travail dépasse les 2.5 millions de lignes de code et est classé comme grand. C'est un logiciel qui exécute des fonctions complexes, communique avec du matériel, possède plusieurs bases de données inter-reliées et a été

écrit en utilisant plus d'un langage de programmation. Dans la section suivante nous allons énumérer les caractéristiques et les fonctions de ce logiciel tout en montrant les difficultés rencontrées lors des tests.

4.3 Difficultés présentes dans le test des grands systèmes

4.3.1 Bases de données inter-reliées

Le logiciel sous test dans ce projet possède plusieurs bases de données inter-reliées. Ces bases de données sont aussi de grande taille. Tester un logiciel complexe exige que les fonctions du logiciel soient vérifiées plusieurs fois. Par exemple, la fonction de modification d'un élément simple pourrait modifier des records dans 10 bases de données reliées. La vérification de la fonction de modification exige 10 vérifications, comme s'il y avait 10 fonctions de modifications indépendantes.

La vérification de la bonne mise à jour des records dans la base de données après un certain test est aussi une tâche difficile. Réorganiser ces bases de données, lorsque de nouvelles fonctions sont ajoutées, peut prendre plusieurs heures ou jours. Le temps nécessaire pour la réorganisation des base de données est aussi une autre mesure de la taille de grands systèmes et de leur complexité. Le testeur devrait s'assurer de ne jamais détruire certaines données lors des tests.

4.3.2 Complexité des fonctions

Un grand logiciel est réalisé pour résoudre des problèmes de grande complexité. Par exemple, notre logiciel exécute des calculs complexes. Il fournit et reçoit des données venant des systèmes externes. Il enregistre et recherche des données dans différentes bases de données par l'exécution d'une fonction simple.

Les logiciels moins complexes ont moins de données, de calcul complexe et d'itérations a faire. Le petit logiciel, offre une réponse déterministe et exige moins de cas de test.

4.3.3 Longue durée de vie

Les grands logiciels sont dispendieux, ce qui exige une durée de vie très longue afin d'amortir leur coût de développement. Tout au long de sa vie, le logiciel subi des modifications du code par des personnes différentes dont chacune utilise sa façon et ses techniques de programmation. Ceci pourrait engendrer des problèmes à cause des différences de méthodes de programmation et de la désuétude de certaines parties du code. En plus, Le concepteur original n'est souvent plus disponible pour répondre aux questions soulevées.

4.3.4 Implication de matériels

Les grands logiciels ont souvent plusieurs interfaces avec le matériel. Tester ces interfaces englobe plusieurs difficultés significatives. Parfois, ces interfaces sont très difficiles à tester ou accéder. Dans ce cas une simulation ou une émulation de ces interfaces pourrait surmonter cette difficulté. Il faut avoir dans l'équipe de test des personnes qui connaissent bien le matériel impliqué. Il ne faut pas tester toutes les interfaces présentes. Le choix des interfaces à tester dépend du type de test à réaliser et des besoins du client.

4.3.5 Criticalité

Les grands logiciels sont ceux qui contrôlent les centrales nucléaires, les lignes de téléphone, le trafic ferroviaire, les navettes spatiales, les équipements de support de vie, le transfert de fonds. Un arrêt ou un mal fonctionnement d'un tel logiciel est souvent

catastrophique pour les utilisateurs. Des questions légales sérieuses peuvent surgir quand un logiciel de cette nature échoue dans ses fonctions.

Selon un article apparu dans le journal IEEE computer [2], pendant le vol du lanceur européen Ariane 5 le 4 juin 1996, un problème informatique a causé l'explosion et la perte totale de la mission 40 secondes après le décollage. Les rapports ont indiqué que le désastre a causé une perte d'un demi milliard de dollars. Cet exemple nous donne une idée de l'énormité des pertes qui pourraient avoir lieu en cas d'échec d'un tel logiciel.

Les logiciels critiques doivent contenir le moins d'erreurs possibles. Les testeurs doivent vérifier le plus de cas possibles et fournir un plan de contingence.

4.3.6 Implications de plusieurs organisations

Dans le cas de notre logiciel sous test, les tests impliquent un grand nombre de personnes géographiquement distribuées avec différentes perspectives et compétences.

4.4 Test complet impossible

Si l'objectif de test est de prouver qu'un programme ne contient pas d'anomalies, alors le test pourrait être non seulement impraticable, mais aussi théoriquement impossible. Il y a différentes approches qui pourraient être utilisées pour démontrer qu'un programme est correct : tests basés sur la structure, tests basés sur la fonction et les preuves formelles d'exactitude. Chaque approche mène à la conclusion qu'un test complet, dans le sens d'une preuve n'est théoriquement ou pratiquement pas possible [24]. Le seul test approfondi est celui qui rend le testeur épuisé [26].

Selon Roger Pressman [17], s'il est possible d'exécuter 1,000 tests par seconde sur un programme de quelques centaines d'instructions, cela prendra 3,000 années pour le tester complètement et exhaustivement. Il s'agit là d'un problème de type NP [25].

Selon Cim Kaner [27] il est impossible de faire un test complet, pour les raisons suivantes :

1- Nous ne pouvons pas tester toutes les entrées au programme :
   Le nombre d'entrées que nous pouvons donner à un programme est typiquement infini. Nous employons différentes techniques pour choisir un nombre restreint de cas de test que nous utiliserons pour représenter l'espace maximal des tests possibles. Le petit échantillon pourrait ou ne pourrait pas trouver toutes les anomalies reliées aux entrées.

2- Nous ne pouvons pas tester toutes les combinaisons des entrées au programme, il s'agit là d'un problème d'explosion combinatoire [25] :
   Supposez qu'un programme nous permet d'ajouter deux nombres. La conception du programme permet au premier nombre d'être compris entre 1 et 100 et le deuxième nombre entre 1 et 25. Le nombre total de paires que nous pouvons tester (ne comptant pas toutes les paires qui utilisent les entrées incorrectes) est 100 x 25 (2500).

3- Nous ne pouvons pas tester toutes les voies d'accès dans le programme. Pour illustrer ceci prenons l'exemple de Glenford Myers [4] :

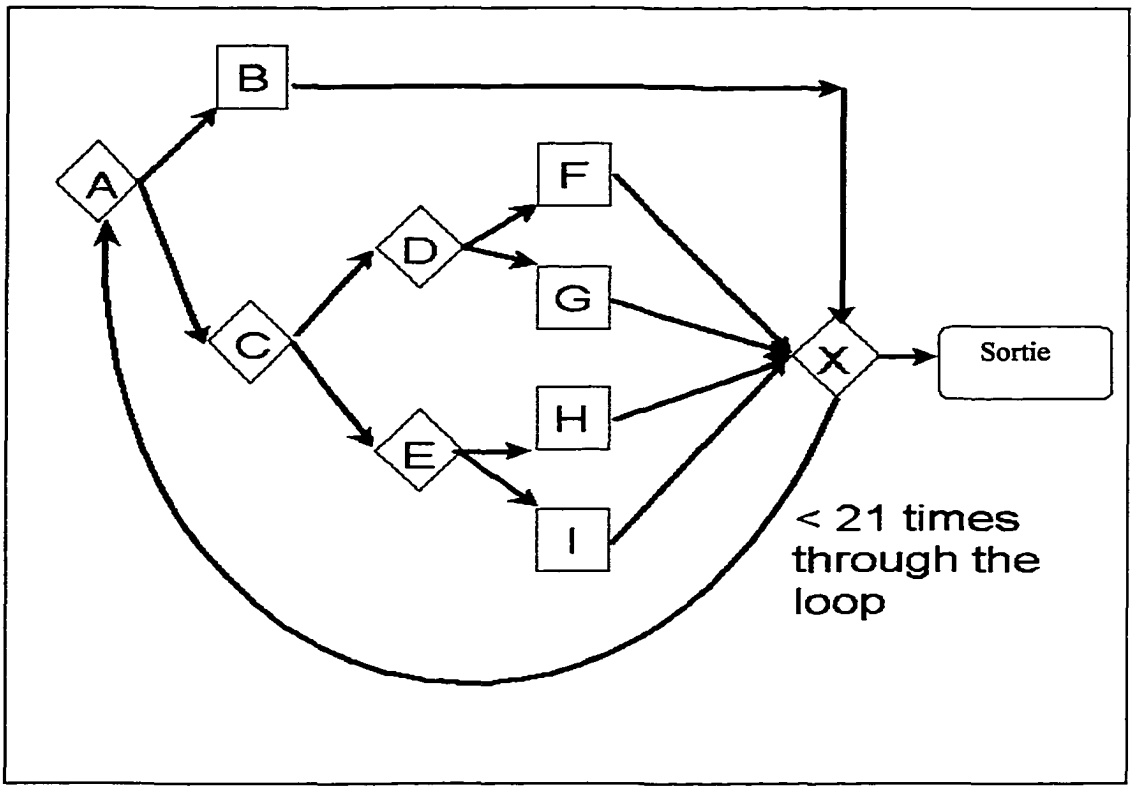


Figure 4.1 Exemple de Myers [4]

Le programme commence au point A et termine à la sortie. On quitte le programme au point X. Quand on est rendu à X, on peut quitter ou faire une boucle de retour à A. On ne peut pas faire une boucle de retour à A plus de 19 fois. La vingtième fois qu'on atteint X, on doit quitter.

Il y a cinq voies d'accès de A à X. On peut aller de A à B à X (ABX) ou on peut aller ACDFX ou ACDGX ou ACEHX ou ACEIX. Il y a ainsi 5 voies d'aller de A à la sortie, si on traverse X seulement une fois.

Si on arrive à X la première fois, et on fait une boucle de retour de X à A, alors il y a cinq voies (les mêmes cinq) d'aller de A de nouveau à X. Ainsi on a 5 x 5 voies d'aller de A à X et puis de A à X à nouveau. Il y a 25 voies d'accès dans le programme qui passeront par X deux fois avant qu'on passe à la sortie.

Il y a 53 voies d'aller à la sortie après avoir passé par X trois fois, et 520 voies d'aller à la sortie après avoir passé par X vingt fois.

Au total, il y a $5 + 5^2 + 5^3 + 5^4 + \ldots + 5^{20} = 10^{14}$ (100 trillions) voies d'accès à travers ce programme. Si on pouvait écrire, exécuter, et vérifier un cas de test à toutes les cinq minutes, on finira de tester dans quelques milliards d'années.

4.5 Solution du problème

La preuve que le logiciel fonctionne ou la preuve que le code a très peu d'anomalies, doit être un but secondaire. En premier lieu, nous devons chercher les anomalies. Deuxièmement, on doit essayer de tester autant de situations différentes que le code le permet. Dans le monde complexe qu'est le développement de logiciel, aucun travail de test ne peut être 100% complet. Considérez juste l'exemple classique de tester chaque entrée possible dans une zone de 10 caractères. En ne considérant que les 26 lettres, on doit exécuter plus de 141167095653400 cas de test ! En résumé, on ne peut jamais démontrer la détection de toutes les anomalies qu'un programme pourrait probablement avoir.

Les testeurs ont créé plusieurs méthodes (morcellement par équivalence, analyse de valeurs aux bornes, etc.) qu'on a décrit dans les chapitres précédents. Ces méthodes servent à couvrir le plus de cas possibles dans un temps raisonnable. Malgré l'efficacité de ces méthodes, elles ne sont pas fiables pour nous aider à bien tester notre logiciel. Le logiciel sous test dans ce travail dépasse les 2.5 millions de lignes de code, ce qui rend le test extrêmement long ou presque impossible à réaliser. En plus, il s'agit d'un logiciel qui

contrôle des fonctions critiques. Une anomalie dans le code pourra coûter chèr en terme de vie et d'argent. Nous avons besoin d'une méthode de test qui nous permettra de couvrir encore plus de cas de test dans le minimum de temps.

Pour un grand logiciel, une méthode est efficace si elle peut orienter notre test et ne tester que les parties du code susceptibles d'anomalies. Dans le cas de notre test par exemple, nous allons chercher à localiser les fonctions où il y a un traitement sur la finitude de grandeurs dans le logiciel. Pour ce faire, il faut utiliser l'une des méthodes statiques soit la revue du code en groupe assisté par ordinateur.

Une fois la revue du code terminée. L'inspection de code de 2.5 millions de lignes est alors limitée à quelques milliers de lignes de code. Ensuite on passe à la phase de création de cas de test. Ces derniers sont établis en se basant sur les parties du code identifiées lors de l'inspection et non pas sur le code au complet. Ensuite, nous allons appliquer nos cas de test en utilisant l'une des méthodes dynamiques, soit la méthode de boîte noire. Mais ce n'est pas la méthode de boîte noire classique que nous allons utiliser, nos tests seront guidés par les résultats de l'analyse du code. La procédure complète de cette méthode est décrite au chapitre 6.

4.6 Conclusion

Le test des grands logiciels renferme beaucoup plus de difficultés que le test de petits ou de moyens logiciels. Ces difficultés proviennent de la complexité des ses fonctions, de ses bases de données inter–reliées et de ses communications avec le matériel. De plus, nous avons montré qu'un test complet, dans le sens d'une preuve est théoriquement impossible et pratiquement infaisable. Nous avons ensuite terminé le chapitre en donnant un bref aperçu de notre méthode de test qui permettra la solution des problèmes reliés au test de grands logiciels. Cette méthode, sera l'objet d'étude du chapitre 6.

CHAPITRE 5

DÉVELOPPEMENT D'UN DICTIONNAIRE CONTEXTUEL

Telle que décrit dans le chapitre précédent, la méthode de test utilisée dans ce travail contient une phase de test statique qui consiste à inspecter le code dans un environnement de groupe assisté par ordinateur. Cette inspection est faite dans le but d'identifier les parties du code qui contiennent ou qui sont en relation avec le type d'anomalie recherchée. Pour ce faire, l'équipe doit bâtir un dictionnaire contextuel basé sur le type d'anomalie à débusquer. Dans ce chapitre, nous allons décrire la démarche à suivre pour créer ce dictionnaire contextuel. Ce dernier va contenir tous les mots clés concernant le problème relatif aux anomalies de la finitude de grandeurs.

Avant d'expliquer la construction du dictionnaire contextuel, nous allons discuter du type d'anomalie que nous cherchons à débusquer. C'est-à-dire, les anomalies logiciels relatives aux problèmes de la finitude des grandeurs. Nous exposerons ce problème en détail et montrerons les systèmes qui peuvent causer des problèmes. Ensuite nous dresserons et analyserons les différentes causes qui ont créé ce problème.

5.1 Définition du problème

La plupart des systèmes informatiques, des logiciels et un grand nombre de composants électroniques gèrent ou emmagasinent des dates. Dans les années cinquante et soixante, la mémoire RAM étant extrêmement rare et coûteuse (et encombrante), les années furent codées sur deux octets (15/03/58, par exemple). De bonne foi, les

programmeurs pensaient que les progrès de la technologie allaient en quelques années rendre obsolète la plupart de leurs développements. Il n'en fut malheureusement pas ainsi. Les logiciels évoluèrent, les lignes de code venant s'ajouter aux lignes de code déjà existantes (pour atteindre parfois plusieurs centaines de millions).

Si les programmes se contentaient de l'impression de ces deux chiffres, ça ne serait pas bien grave. La plupart des programmes font des calculs portant sur ces deux chiffres. Par exemple "98 - 60 = 38". Et se servent du résultat de l'opération pour prendre une décision (vous avez cotisé 38 ans, vous pouvez donc prendre votre retraite). Le même programme faisant le calcul en l'an 2000 va donner : "00 - 60 = -60" et indiquer que "n'ayant jamais cotisé, vous n'avez aucun droit".

Alors comme on vient de voir le problème de la finitude des grandeurs est celui de l'incapacité de gérer une donnée dès qu'elle dépasse une certaine grandeur limite.

La cause première du problème semble faussement simple. Étant donné que les ordinateurs sont programmés pour enregistrer les dates en utilisant les deux derniers chiffres de l'année, la plupart des systèmes informatiques interpréteront le premier janvier 2000 comme étant le 1er janvier 1900. En outre, de nombreux ordinateurs ne sont pas programmés pour tenir compte du fait que l'an 2000 est une année bissextile. Ainsi au passage du nouveau millénaire, de nombreuses applications ont effectué des calculs erronés ou cessé simplement de fonctionner.

Le problème relatif à la finitude de grandeurs ne touche pas seulement les systèmes financiers, mais également tous les systèmes commerciaux et les processus informatiques. La défaillance d'un petit élément dont la logique de traitement dépend des dates peut avoir des conséquences graves. Par exemple, dans le cas d'entités qui fonctionnent à l'intérieur d'un environnement de traitement réparti, une défaillance dans

les communications pourrait avoir des répercussions importantes sur l'ensemble de la structure de la technologie de l'information.

Ce problème affecte presque toutes les entités. Il ne se limite pas aux grandes entreprises qui utilisent des gros processeurs. Il peut toucher des organisations de toutes les tailles, qui utilisent toutes sortes de logiciels qui reposent sur le traitement des données. Ou encore des organisations qui font des affaires avec des fournisseurs clés ou des clients qui utilisent de tels logiciels.

5.2 Domaines touchés par le problème relatif à la finitude de grandeur

Des problèmes peuvent apparaître dans les systèmes suivants (tous essentiels à l'organisation des entreprises) :

- Ordinateurs personnels
- Systèmes informatiques
- Systèmes de gestion des données
- Systèmes de contrôle de l'environnement
- Systèmes de transport
- Systèmes de sécurité
- Systèmes de production de relevés
- Systèmes téléphoniques.
- Services d'utilité publique (la fourniture d'électricité, de services téléphoniques, d'eau et de gaz.)
- Systèmes intégrés
- Systèmes d'alarme
- Équipements d'encodage
- Appareils de climatisation
- Appareils servant à indiquer les dates d'échéance/de péremption
- Télécopieurs
- Systèmes de gestion des horaires
- Ascenseurs/Escaliers mobiles
- Systèmes d'éclairage, de chauffage et de climatisation
- Équipements de surveillance
- Équipements d'imprimerie et d'emballage
- Standards téléphoniques
- Systèmes de vidéoconférence
- Systèmes de Cartes de crédit, et bien d'autres systèmes.

5.3 Causes générales du problème

1. *Manque de normes de datation acceptées à grande échelle*

Nous n'avons pas de représentation normalisé des dates qui est acceptée à l'échelle internationale. Par exemple, pour les Américains, 1/4/96 signifie le 4 janvier, 1996. Pour un Anglais, 1/4/96 signifie le premier avril, 1996. Au moins l'année est commune entre ces deux. Mais même ceci n'a pas été vrai autour du monde - la Chine, par exemple, n'a pas normalisé selon le calendrier grégorien jusqu'à 1948.

Des tentatives ont été faites depuis les années 70 pour fixer de telles normes, et même récemment pendant les années 80 la norme ISO-8601 a préconisé le format de yyyymmdd pour sauvegarder la date. Mais l'utilisation de ces normes n'a pas été exigée à une échelle internationale. Sans une norme largement acceptée, les équipes de gestion et de technologies de l'information font les choix qui leur convienne. Alors, une mauvaise interprétation de certaines dates n'est pas quelque chose de surprenant.

2. *Les applications ont une durée de vie plus longue que prévue*

Les concepteurs des applications ont utilisé des algorithmes de datation qui échoueraient en traitant des dates au-delà du 20^{e} siècle. Ils justifiaient cette utilisation par le fait que l'application a une durée de vie qui ne dépassera pas notre ère.

En outre, une fois l'application en fonction, il y a une hésitation considérable pour la changer de manière significative ou à la remplacer. " S'il n'est pas en panne, ne le changez pas ", a longtemps été un principe non écrit pour beaucoup d'organisations. Ceci semble raisonnable à court terme et permet d'épargner de l'argent. Cependant, à long terme le coût de l'entretien et des modifications du code anéantiront cette économie.

3. *L'absence de prévisibilité*

Bien souvent, malheureusement, l'ensemble des conséquences à long terme d'un choix est impossible à prévoir et ce, quelle que soit la (bonne) volonté et les compétences. Ceci n'est pas propre à la corporation des informaticiens : à titre d'exemple, les inventeurs de l'automobile pouvaient-ils prévoir la pollution et les accidents mortels de la circulation ? Et si oui, devaient-ils pour cela renoncer à cette invention ? De même, Graham Bell, lorsqu'il inventa le téléphone en 1875 à l'Université de Boston, imaginait-il qu'un siècle plus tard son invention serait devenue l'une des plus indispensables à la vie courante ? Et aurait-il pu concevoir que sous sa forme portable, ce même téléphone deviendrait une nouvelle forme de pollution dans les lieux publics ? [3]

4. *La nécessité de la compatibilité*

Il est très souvent nécessaire d'assurer la compatibilité (dite ascendante) des nouveaux programmes avec les anciens. Ainsi se créent des relations de dépendance fortes vis a vis de choix antérieurs [3].

Par exemple, si le logiciel Windows n'avait pas pu exécuter des applications de DOS, Microsoft aurait été mal en partant. Aujourd'hui, Windows95 est confronté au fardeau de supporter non seulement des programmes de DOS mais également Windows 3.1, Windows for Workgroups et ainsi de suite. Ceci signifie que des algorithmes qui ne sont pas à jour ont dû être supportés.

5. *Contraintes sur les ressources informatiques*

La mémoire, l'espace disque et même l'espace de cartes perforées étaient coûteux et rares aux premiers jours de l'informatique. N'importe quel moyen qui pourrait économiser de l'espace et de l'argent a été appliqué.

Dans les premiers jours de l'informatisation, il semblait raisonnable de ne pas inclure de chiffre supplémentaire pour les années. En fait, pour cette même raison, quelques applications ont été écrites avec seulement un chiffre pour l'année. Celles-ci doivent être réécrites pour faire face à leur premier changement de décennie. Maintenant après quelques dizaines d'années ceci est une tâche fort complexe.

6. *Réutilisation de code*

Il a toujours semblé raisonnable économiquement de ne pas réinventer la roue. Pratiquement toutes les nouvelles applications ont des algorithmes et même du code incorporé des systèmes précédents. Ceci accélère le développement et donne (habituellement) des systèmes plus fiables.

Les algorithmes qui ont une erreur cachée de traitement relatif à la finitude de grandeur ont été réutilisés. Il s'agit de l'une des raisons pour laquelle le problème est si énorme. Au fur et à mesure que les algorithmes sont utilisés et réutilisés, leur charge mortelle est distribuée sur de plus en plus de systèmes.

5.4 Effet domino

En effet, aujourd'hui, presque tous les ordinateurs et de nombreux matériels sont intégrés dans des réseaux. De ce fait, leur fonctionnement correct est dépendant d'un certain nombre d'éléments clés du réseau. Des ordinateurs de services (serveurs, pare-feu, routeurs, commutateurs, etc.) tous utilisant des horloges sont tous susceptibles tomber en panne. Le risque ici est celui de pannes "en cascade" ou d'effet domino. Dans ce type de panne, un arrêt d'un élément à cause d'une certaine panne provoque une suite de mal fonctionnements conduisant à l'arrêt de tous les autres éléments dans le réseau mêmes s'ils ne sont pas en pannes.

5.5 Développement d'un dictionnaire contextuel

Le dictionnaire contextuel constitue un élément principal de notre méthode de test. Il doit contenir les mots clés nécessaires à l'identification des parties du code en relation avec l'anomalie recherchée. En général, il est créé par un groupe d'expert dans le langage de programmation inspecté et l'anomalie recherchée.

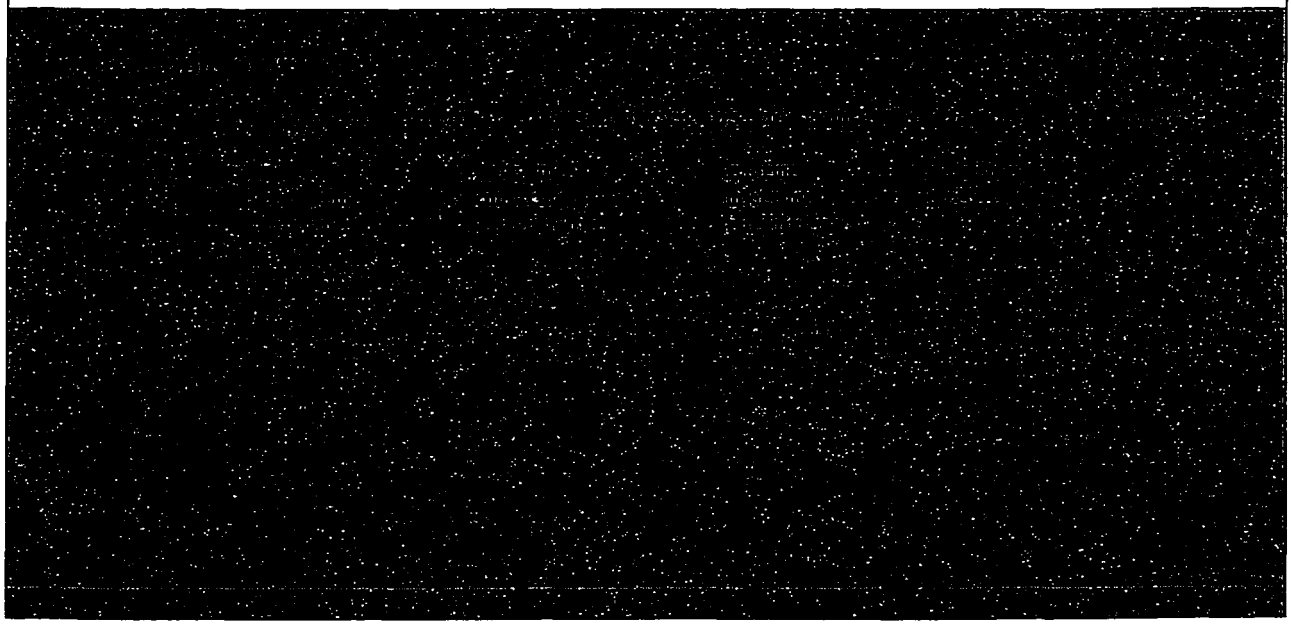
Dans le cadre de ce travail, le dictionnaire contextuel devrait contenir tous les mots clés qui indiquent la présence d'un traitement de la finitude de grandeur. Le choix des mots clés est aussi basé sur le langage de programmation. Dans notre application le langage utilisé est le langage C. Sur les pages suivantes on trouve la liste des mots clés contenus dans le dictionnaire créé et utilisé dans ce travail.

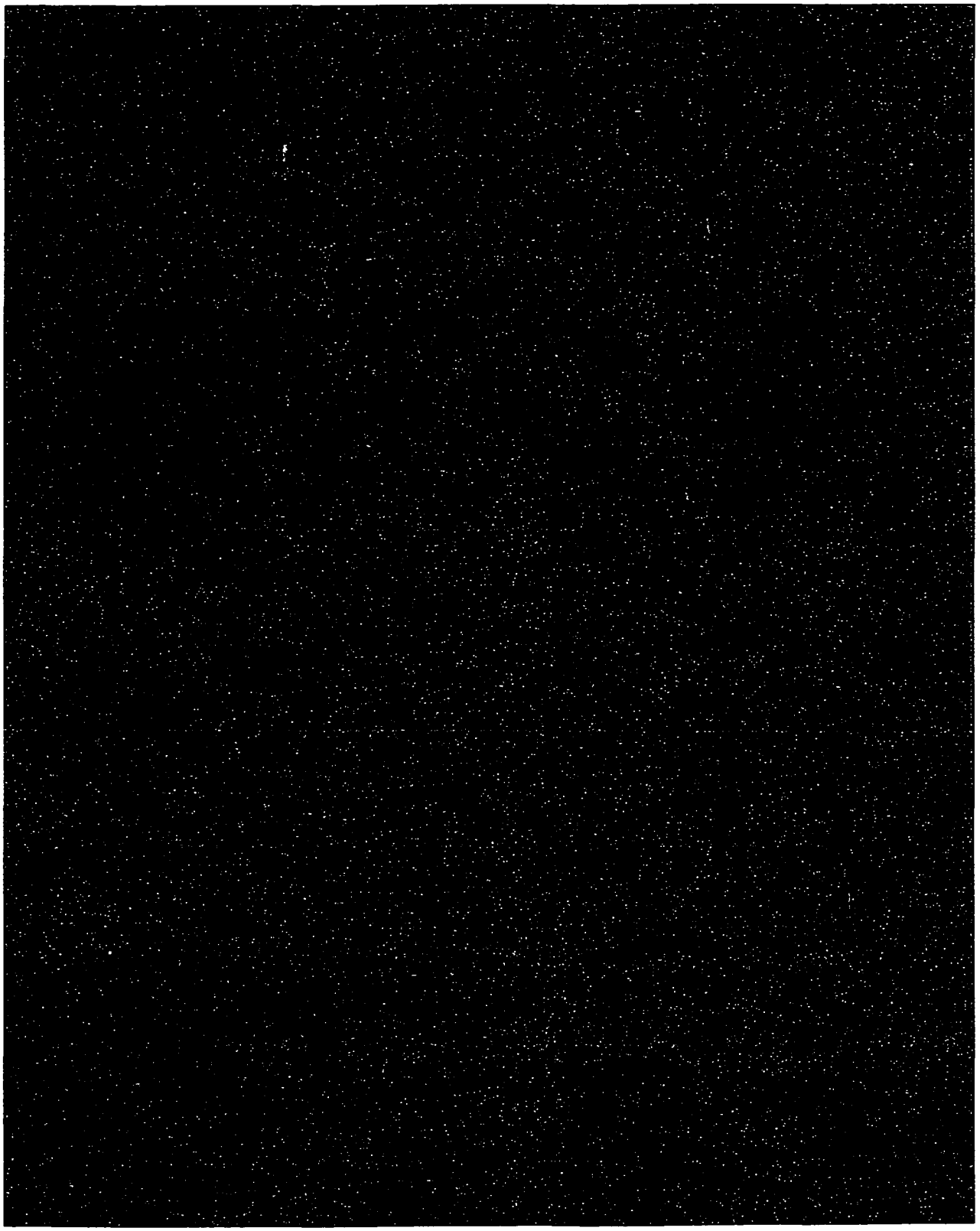
1. 0x70
L'interruption de l'horloge en temps réel (real time clock, RTC) en hexadecimal.

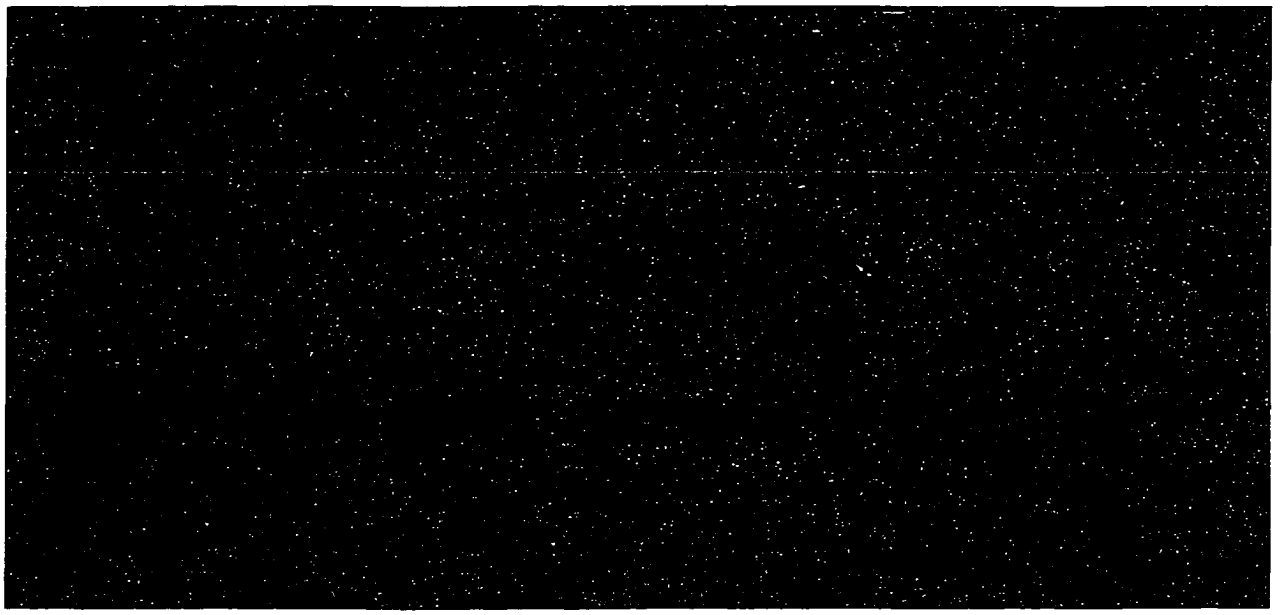
2. IRQ : Ligne des demandes d'interruption (interrupt request line)
Une ligne matérielle sur laquelle un dispositif tel qu'un port entrée/sortie, un clavier ou un lecteur de disques, peut envoyer des demandes d'interruption à l'unité centrale de traitement (UCT). Les lignes des demandes d'interruption sont établies dans le matériel interne de l'ordinateur et ont chacune un niveaux de priorité de sorte que l'UCT pourra déterminer les sources et l'importance relative des demandes d'interruption.

3. time.h
Une bibliothèque contenant les fonctions de traitement de la date et de l'heure.

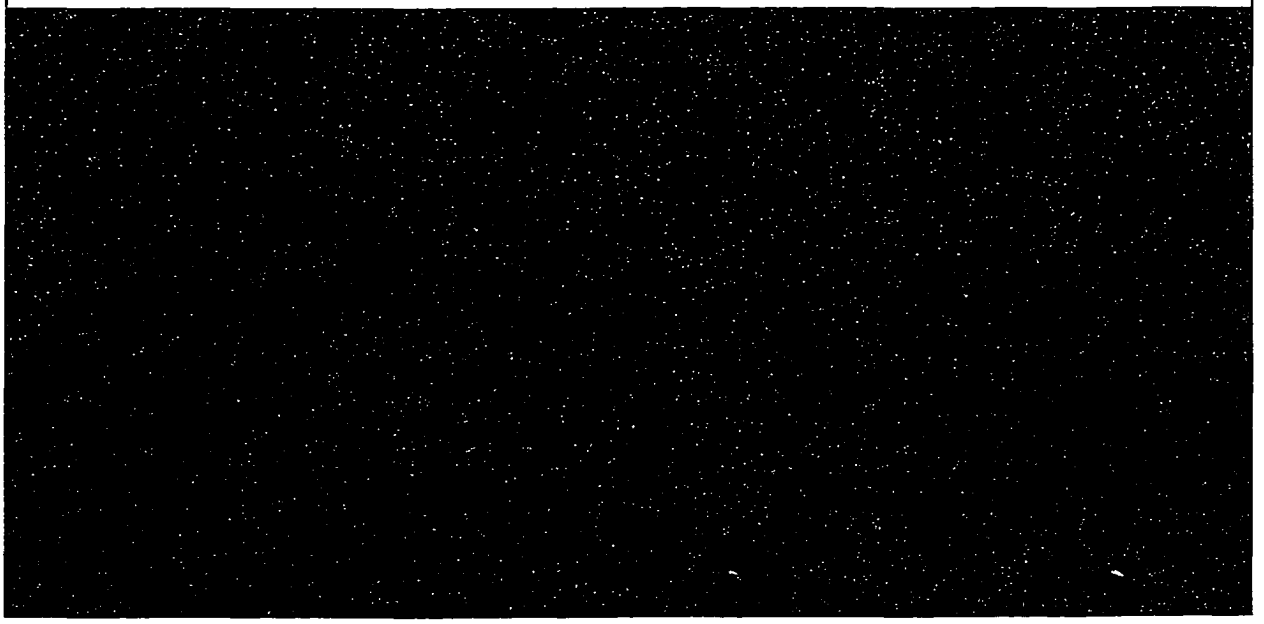
4. time
La fonction de temps renvoie le nombre de secondes écoulées depuis minuit (00:00:00), janvier 1, 1970, temps universel, selon l'horloge du système.

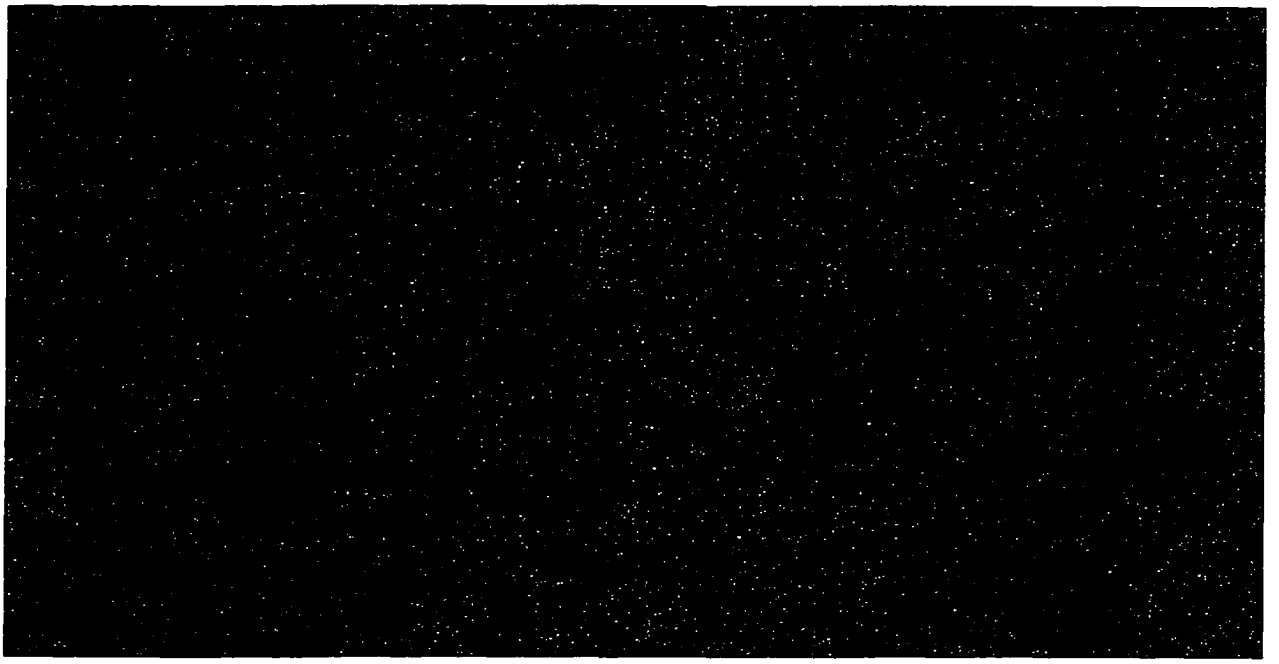






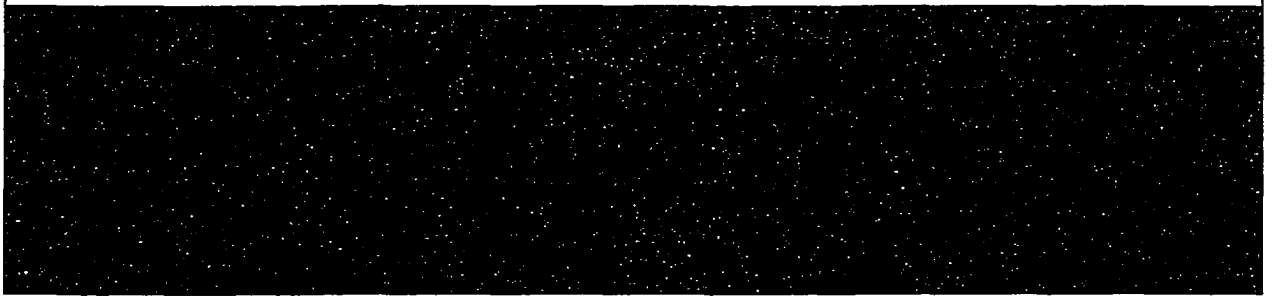
5. Clock, CLOCKS_PER_SEC
Il calcule le temps utilisé par le processeur pour le processus appelé. L'horloge renvoie le nombre de coups d'horloge de temps écoulé du processeur. La valeur retournée est le produit de la période de temps qui s'est écoulé depuis le début d'un processus et la valeur de la constante CLOCKS_PER_SEC. Si la période de temps écoulée n'est pas disponible, la fonction renvoie –1.

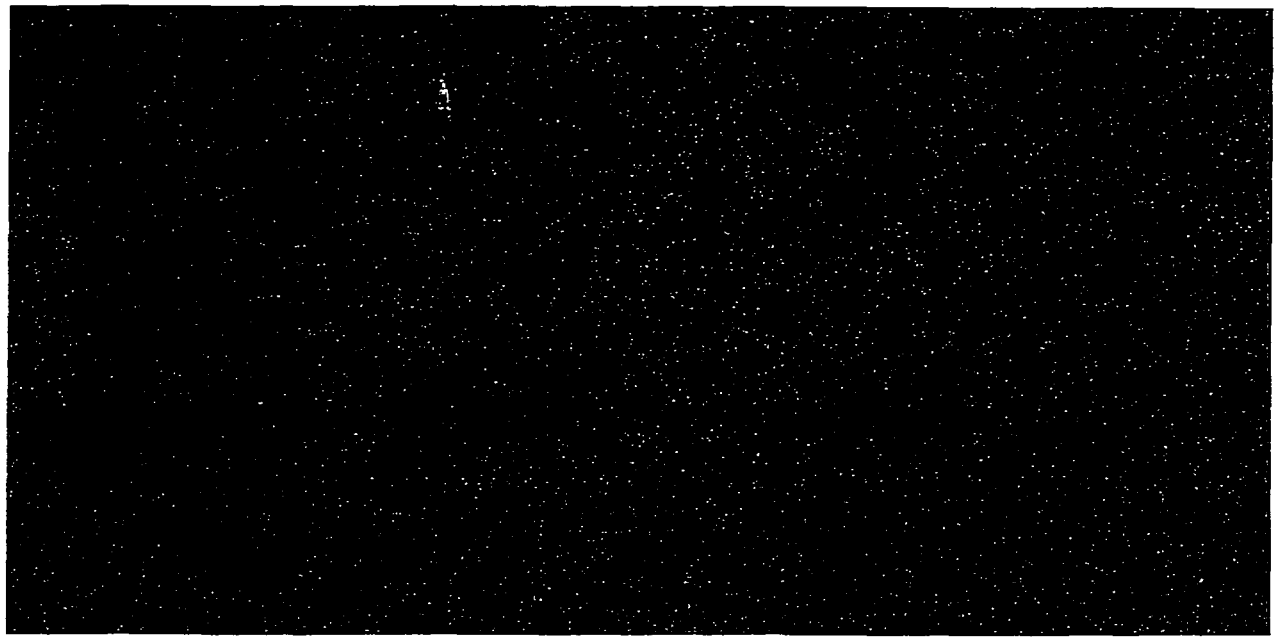




6. tm
Il est utilisé par asctime, gmtime, localtime, mktime pour sauver et retrouver les informations concernant le temps

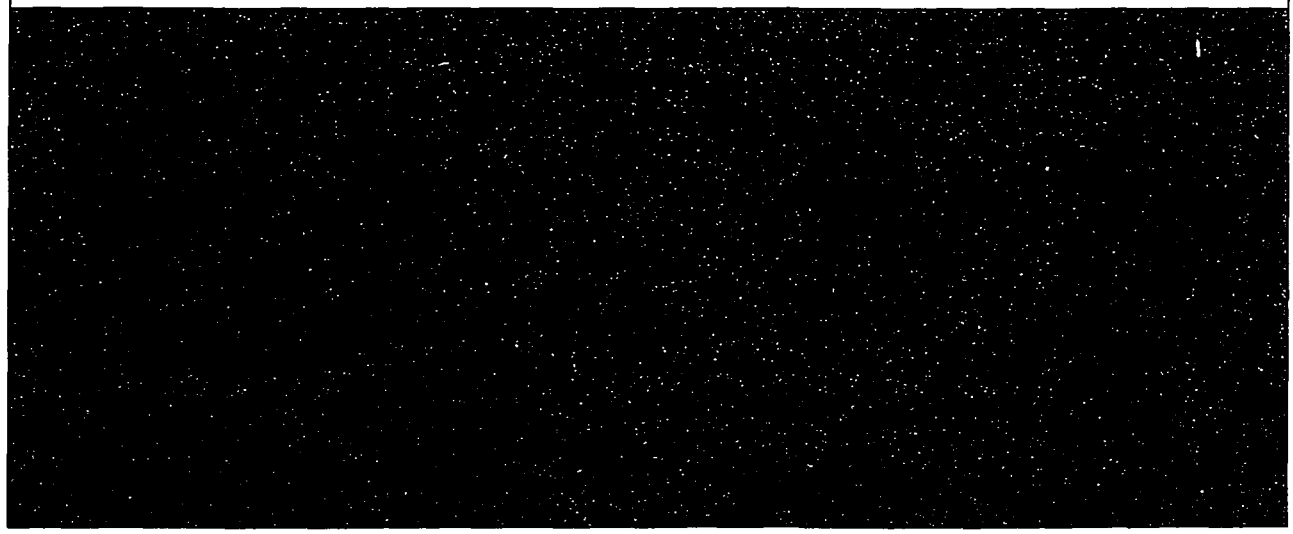
7. _tzset
La fonction _tzset utilise la configuration actuelle de la variable de l'environnement TZ pour assigner des valeurs à trois variables globales: _daylight, _timezone et _tzname. Ces variables sont utilisées par les fonctions de _ftime et de localtime pour faire des corrections à partir du temps universel (UTC) à l'heure locale, et par la fonction time pour calculer l'UTC à partir du temps de système.





8. _timezone , _daylight, et _tzname
Ils sont utilisés dans quelques routines de la date et l'heure pour faire des ajustements sur le temps local. Ils sont déclarés dans TIME.H tels : extern int _daylight; extern long _timezone; extern char *_tzname[2];

9. _ftime
Il donne le temps actuel



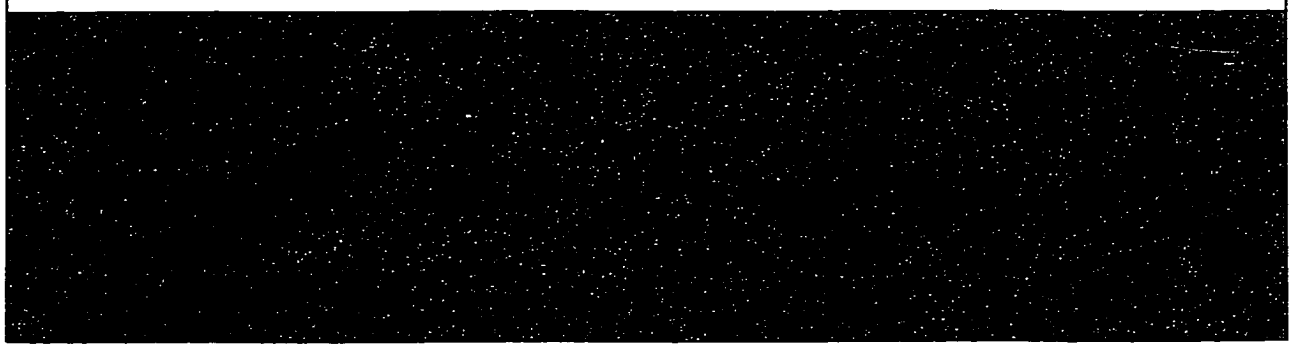
[redacted]

10. CLK_TCK
La même fonction que CLOCKS_PER_SEC, mais considérée comme désuete.

11. Difftime
Il trouve la différence entre deux temps.

[redacted]

12. __asm, __pascal, __fortran, __ada
Ces mots clés sont utilisés pour indiquer l'insertion du code d'un autre langage de programmation.



En plus de ces mots clés qui indiquent la présence directe d'un traitement relatif à la finitude des grandeurs. Il faut inclure dans le dictionnaire contextuel tous les autres mots qui indiquent la présence indirecte d'un traitement relatif à la finitude de grandeurs. En d'autres termes les mots reliés aux problèmes de traitement de la finitude des grandeurs. Nous avons classé ces mots en cinq catégories :

1. *Les mots clés concernant la date :*

a. Les intervalles de dates : 19, 20, 99
b. century
c. date (à l'exception des dates de création de programme ou de versions et les mots update ou validate)
d. day
e. leap
f. month
g. year

2. *Les mots clés concernant les formats de datation :*

dd/mm/yy; dd/mm/yyyy; ddmmyy; ddmmyyyy; mm/dd/yy; mm/dd/yyyy; mmddyy; mmddyyyy; yy; yy/mm/dd; yymmdd; yyyy; mm; yyyy/mm/dd; yyyymmdd; yyyy-mm-dd; yyyy.mm.dd; ddmmyyyy

3. *Les mots clés concernant le temps :*

hour; hr (et tout ce qui pourrait suivre comme hrs, hr:, hr; Etc.); min(et tout ce qui pourrait suivre comme min:, minu, min;, etc); msec;sec (sauf section); timeout; time; timer; timing; tick; tzset; timezone; daylight; tzname; TZ; clock; clk; CLK_TCK

4. *Les mots clés concernant les interruptions:*

L'interruption de l'horloge en temps réel (Real Time Clock) en Hexadecimal (0x70); interrupt; irq

5. *Les mots clés reliés au temps ou à la date :*

password; gps ; delay; stamp; sync; rtc

5.6 Conclusion

L'objectif de ce chapitre est de montrer comment établir un dictionnaire contextuel qui est l'élément essentiel de notre méthode de test.

Au début de ce chapitre nous avons défini le type et le domaine des anomalies que nous cherchons à débusquer. Ensuite nous avons dressé les causes qui ont mené à ce type d'anomalies. Puis nous avons donné une série d'exemples qui présentent les conséquences fâcheuses causées par ce type d'anomalies.

Nous avons terminé ce chapitre en montrant comment développer un dictionnaire contextuel. Nous avons dressé une liste des mots-clés de ce dictionnaire, donné leur explication et illustré avec des exemples.

Dans le chapitre suivant, nous allons voir en détail notre méthode de test pour les grands logiciels. Cette méthode est très bien adaptée pour les établissements spécialisés dans le test des logiciels.

CHAPITRE 6

MÉTHODE DE TEST

La revue de code englobe un large éventail d'activités. De la revue informelle individuelle à la revue effectuée par groupes assistés par ordinateur en utilisant les modèles de processus intégrés. Il existe actuellement trois types principaux de réunion de révision : Inspection, *Walkthrough* et revue technique formelle (FTR). Voici la définition de chacun de ces types :

- Une inspection permet de détecter et résoudre des erreurs le plus tôt possible lors du processus de développement de logiciel et assure que les erreurs ne sont pas propagées d'une phase de développement à l'autre [28]. Elle est une rencontre structurée durant laquelle du personnel technique analyse un artefact de façon systématique afin de produire un artefact de meilleure qualité [29].

- Un *walkthrough* est considéré comme étant une revue informelle effectuée par des pairs où il n'est pas nécessaire de se préparer et les suivis de corrections sont inexistants [30].

- Une revue technique formelle (FTR) est une méthode impliquant un mélange structuré dans lequel un groupe de personnel technique analyse un artefact en utilisant un processus bien spécifié. Le résultat est un autre artefact structuré qui évalue ou améliore la qualité de l'artefact initial aussi bien que la qualité de la méthode[29].

Dans ce chapitre, nous introduisons une méthode de test basée sur la revue du code pour la création des cas de test. Cette méthode utilise surtout la procédure du walkthough mais englobe aussi des caractéristiques tirées des deux autres méthodes que l'on vient de définir.

6.1 Définition de la méthode du code *walkthrough*

Le code *walkthrough* utilise un ensemble de procédures et de techniques de détection d'erreurs pour la lecture du code en groupe. Pour que le *walkthrough* soit efficace, l'équipe doit visualiser la revue comme un outil pour améliorer la qualité des produits de l'équipe. L'équipe de *walkthrough* est composée de trois à cinq programmeurs. Les suggestions pour les participants incluent :

1. Au moins un programmeur fortement expérimenté pour chaque langage de programmation utilisé dans le code et ayant un niveau de connaissance élevé de la procédure et de la méthode utilisée.
2. Au moins une personne qui connaît le matériel communiquant avec le logiciel sous inspection et les protocoles de communication. En plus, de connaître le langage de programmation du code sous inspection.
3. Un nouveau programmeur (pour donner un point de vue nouveau et impartial et pour faciliter la croissance de cette personne comme programmeur).
4. Un programmeur expérimenté avec le langage du code du logiciel sous test, ayant des bonnes relations avec les ressources humaines disponibles dans les autres départements.

6.2 But et objectifs du *walkthrough*

Le but du *walkthough* inclut dans notre méthode est d'accélérer le processus de création de cas de test et de rendre les cas de test plus efficaces. En même temps, il aide à trouver et énumérer les erreurs dans le logiciel. Aucune tentative ne devrait être faite lors de la réunion pour corriger les erreurs trouvées, bien que les suggestions pour des corrections soient certainement appropriées.

Voici une liste des premiers objectifs du *walkthrough* :

a. Créer des cas de test en se basant sur la revue du code;
b. Améliorer la lisibilité et l'entretien du code source du logiciel;
c. Améliorer la qualité et l'efficacité des algorithmes;
d. Vérifier l'exactitude des algorithmes;
e. Rendre le logiciel plus robuste;
f. Trouver les défauts de logiciel – particulièrement les anomalies cachées ou latentes;
g. Vérifier que le code se conforme aux normes;
h. Vérifier que le code répond aux exigences;
i. Vérifier que le code et le manuel d'utilisateur conviennent;
j. Vérifier que le logiciel correspond exactement à la documentation associée.

6.3 Procédure du code *walkthrough* automatisée fait dans un environnement de groupe

Dans ce travail le logiciel sous test est fait par une autre organisation indépendante et géographiquement éloignée de l'équipe de test. Ce genre de situation est difficile et renferme beaucoup de complications pour une organisation de test. De plus cette situation est très répandue de nos jours. Les organisations qui conçoivent des logiciels préfèrent faire tester leurs produits par des organisations indépendantes. Un des objectifs de ce mémoire est de donner une méthode de test pour ce genre de situation difficile et de la détailler de façon complète. Cette méthode a pour but de faciliter la tâche

de telles organisations, leur permettre d'épargner du temps, avoir des communications plus efficaces et d'effectuer des tests plus poussés.

Dans cette section nous décrirons le processus complet de cette méthode baptisée « procédure pour du code *Walkthrough* automatisé fait dans un environnement de groupe »

Demander à l'organisation mère que chaque programmeur de code révise les points suivants dans son code source, avant de soumettre le code pour la revue :

a. Le code est bien commenté.

b. Les commentaires et le code sont clairs.

c. Recompiler, retester l'application et corriger toutes les erreurs trouvées.

d. Le code est écrit selon des normes.

Planifier la date et la façon de l'envoi du code source à partir de l'établissement d'origine à l'établissement où le code *walkthrough* va se faire. En plus, il est préférable de se mettre d'accord sur le format et le moyen de sauvegarde du code (tape, disquette, CD, etc.).

Produire un échéancier précis, spécifier le site où le *walkthrough* va se faire et surtout se mettre d'accord sur les dates limites afin d'éviter toutes interruptions.

Recueillir et échanger les coordonnées, les disponibilités, les décalages horaires et le meilleur moyen de communication (vidéo conférence, courrier électronique, message board, groupe de discussion privée, conférence téléphonique, etc.)

Choisir deux à cinq programmeurs (idéalement quatre) familiers avec le langage du code. Au moins un à deux (selon le cas) de ces programmeurs devraient être à un niveau d'expert et devraient être au courant des concepts de qualité, des méthodes de programmation structurées et des normes de l'organisation.

Diviser le code source en plusieurs sous-ensembles pour ensuite les faire passer en revue par les différents membres de l'équipe. Dans le cas de très grands logiciels, le code source pourrait contenir plus d'un langage de programmation. La division du code se fait selon l'expertise de chacun des membres de l'équipe.

Média et format :

a. Dans le cas de *walkthrough* manuel du code : L'impression sur papier est préférée pour des revues en groupe. Parce que chaque réviseur peut faire des notes pendant que les idées se produisent et il sera plus facile pour un groupe d'insérer des commentaires et de marquer le code.

b. Dans le cas de *walkthrough* automatique du code : En utilisant un logiciel comme « Wingrep » (http://web.pncl.co.uk/~huw/grep/) ou autre, les parties du code qu'on cherche sont marquées et on pourrait les transporter dans un autre logiciel ou éditeur de texte comme « Excel » ou « Word » par exemple pour documenter, faire des statistiques, générer des graphiques, etc. On pourrait aussi utiliser d'autres logiciels faits uniquement pour documenter les résultats du code *Walkthrough*, comme le logiciel « *Walkthrough* Documenter ».

Faire les préparations suivantes pour le code *walkthrough* :

Chaque réviseur devrait avoir les documents, les accès et les outils suivants avant le début du processus du *walkthrough* :

a. L'accès au serveur où se trouve le code source.
b. Les documents décrivant le processus et les objectifs de la revue du code.
c. Les manuels d'utilisation du logiciel pour l'utilisateur et l'administrateur.
d. Les documents et les graphes décrivant les relations du logiciel avec les autres systèmes, les normes et les protocoles de communication.
e. Les manuels sur les simulateurs et les bancs de test utilisés.
f. Copie en format électronique ou sur papier de toutes copies d'écran associées.
g. Toute documentation de conception utile, des manuels ou toute autre source d'information pertinente.
h. Une copie de l'outil permettant de faire du code *Walkthrough* automatisée.

Familiariser les membres de l'équipe avec le logiciel en leur fournissant une session d'information donnée par un expert. Ces sessions pourraient être faites sur place avec le vrai logiciel, par vidéo conférence, en utilisant un émulateur ou par conférence téléphonique en ayant des copies d'écran en main.

Mettre une copie du fichier contenant les mots clés du dictionnaire contextuel et les explications pertinentes dans le répertoire du projet sur le serveur. Tous les membres de l'équipe devront y avoir accès.

Faire réviser le code par chaque membre de l'équipe, lui demander d'insérer ses commentaires et suggestions et documenter les résultats. Dans ce projet, cette étape est réalisée dans un environnement de travail de groupe assisté par l'outil d'automatisation approprié.

Produire ou mettre à jour les trois documents suivants à la fin de chaque période de code *Walkthrough* :

a. Un document contenant les parties du code traité, les mots clés marqués et les statistiques du *Walkthrough* obtenues dans cette période.
b. Un document contenant les commentaires sur les sections du code qui peuvent générer des cas de test.
c. Un document contenant les questions à poser aux concepteurs.

Deux réunions entre les membres de l'équipe devraient être faites après chaque période de *walkthrough*. Le déroulement de ces réunions est décrit dans la section 6.5.1.

Une réunion ou plus devrait être faite avec le personnel de l'assurance de la qualité.

6.4 Organigramme de la méthode de test

Dans cette section, nous montrons l'organigramme de la méthode au complet suivi d'une description des différentes phases.

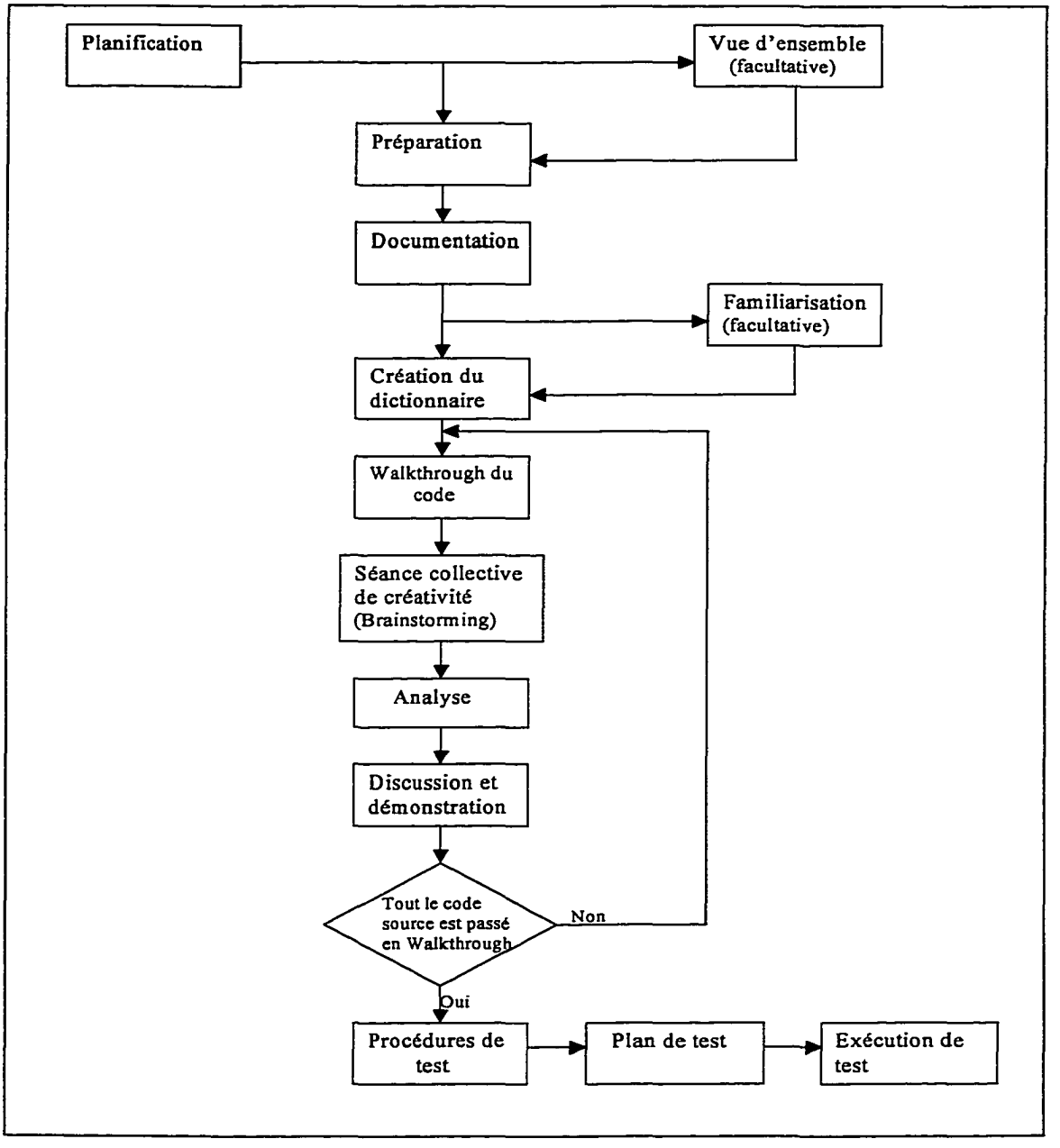


Figure 6.1 Organigramme du processus complet de la méthode de test

1- Planification : Formation de l'équipe, assignation des rôles, choix de l'outil et établissement d'un échéancier.

2- Vue d'ensemble : Ceci est une étape facultative pour donner une vue globale du plan à tous les membres de l'équipe.

3- Préparation : S'assurer que le code est commenté, les commentaires et le code sont clairs, le code a été recompilé et retesté avant de le soumettre et se mettre d'accord sur le format de l'envoi. En plus, il faut échanger les coordonnées et les disponibilités, sur le moyen de communication et sur toute autre activité de préparation.

4- Documentation : Apporter toute la documentation disponible sur le logiciel et créer une base de données contenant le code source, les documents, l'outil d'automatisation, le processus, etc.

5- Familiarisation : Une séance de familiarisation pour l'équipe de test avec le logiciel donnée sur le site sera très utile mais pas obligatoire.

6- Création du dictionnaire : Créer un dictionnaire contextuel (voir chapitre 5) qui va nous aider à identifier les parties du code en relation avec l'anomalie recherché pour ensuite bâtir des cas de test fonctionnel.

7- *Walkthrough* du code : Passer le code en revue et produire les trois documents nécessaires qui vont aider à la création des cas de test.

8- *Brainstorming* : Une première version des cas de test est sortie à cette étape.

9- Analyse : Les cas de test sont analysés individuellement. Un document contenant les cas de test, les questions et les remarques doivent être générés individuellement.

10- Discussion et démonstration : Tous les cas de test seront discutés. Leur utilité et nécessité seront démontrées. Un document final contenant tous les cas de tests finaux sera produit à ce stade-ci. Ces cas de test sont des tests fonctionnels (morcellement par équivalence, analyse de valeurs aux bornes, etc.)

11- Tout le code est passé en *walkthrough* : Puisque le code source était divisé en plusieurs parties. Cette vérification assure que tout le code source est passé en revue.

12- Procédures de test : Écriture des procédures de test.

13- Plan de test : Élaboration du plan de test.

14- Exécution de test : L'équipe exécutera les tests et produira un document contenant les résultats.

6.5 Réunion de révision dans le processus

Après la phase du code *Walkthrough*, les réviseurs doivent se réunir pour valider, échanger des informations et s'informer sur la progression de la procédure. Cette réunion s'appelle une réunion de révision technique (RRT) qui consiste à vérifier l'état courant du projet et de valider les spécifications des tâches subséquentes. Une RRT requiert la présence de plusieurs réviseurs pour une certaine période de temps et doit se dérouler selon des procédures établies [31].

Une étude avait indiqué qu'une réunion de révision effectuée en une seule étape est la méthode d'inspection la plus efficace. Mais elle était basée sur des comparaisons et non sur les caractéristiques intrinsèques de ce type de réunion. Il est maintenant important de tenter de comprendre les activités retrouvées lors de ces réunions afin de les améliorer [32].

Les différentes activités qui se déroulent lors d'une réunion de revue technique[33] sont :

Activités de présentation :

La progression de la réunion est gérée par la présentation des avancements de travaux. L'activité de présentation identifie tous les épisodes où un participant à la réunion présentera son travail afin de le soumettre à l'évaluation des autres participants.

Activités de demande :

La demande est une activité où un participant sollicite un comportement précis (question, explication, etc.) de la part d'un autre participant à la réunion. Ce comportement étant spécifié par l'objet de cette demande. La demande sera donc spécifiée par une activité afin de la préciser.

Activités de gestion :

Les activités de gestion sont les actions qui déterminent le comportement que doivent prendre les participants à une réunion. Il existe deux types d'activités de gestion : la coordination et la planification. Les activités de gestion se réfèrent aux besoins de s'assurer que la réunion suit son cours ou qu'une certaine activité peut avoir de l'impact sur le projet.

Activités de discussion :

La discussion est le groupe d'activités principales d'une réunion de révision technique. Ce groupe est divisé en dix activités qui permettent d'identifier très précisément un épisode particulier :

1. **Justifier**: Une justification est l'action d'argumenter ou de prouver le bien-fondé du pourquoi d'un certain choix. Il est souvent nécessaire de répondre à des évaluations par une justification de l'approche prise. On peut justifier plusieurs artefacts et tâches.

2. **Informer**: Action de porter des renseignements à la connaissance d'une ou de plusieurs personnes au sujet de la nature d'une solution ou d'un critère.
3. **Évaluer**: Action de porter un jugement sur la valeur d'un objet. On émet une opinion sur un fait quelconque. Cette évaluation peut être négative, positive ou neutre.
4. **Accepter**: Action de considérer un artefact ou une tâche comme valide.
5. **Rejeter**: Refuser ou écarter un artefact ou une tâche comme non valide.
6. **Développer**: Action d'exposer en détail une idée nouvelle au sujet d'une solution ou d'un critère. Il s'agit donc d'une activité créatrice.
7. **Faire une hypothèse**: Présenter la représentation que l'on se fait d'un cas ou d'un critère. Cette représentation a le statut cognitif d'hypothèse ce qui s'exprime verbalement par des expressions comme "je crois que", "je pense que...", "...peut-être".

6.6 Phase de création de cas de test

Une fois le code *Walkthrough* terminé, les trois documents sont produits, l'équipe passe à la phase la plus importante du projet, qui est la phase de création de cas de test. Le degré de succès de cette phase dépend des facteurs suivants :

1. Le bon choix des mots clés dans le dictionnaire contextuel ;
2. Le document contenant les parties du code identifiées, les mots clés marqués et les statistiques sur le code, ont été produit correctement lors de la phase *Walkthrough*;
3. Les questions relevées lors de la phase du code *Walkthrough* ont été répondues par les concepteurs et documentées.

Alors, on voit bien que les étapes précédentes constituent la base nécessaire pour créer les cas de test et par la suite construire les procédures de test. C'est pourquoi il faut exécuter ces étapes précédentes avec beaucoup de précision et d'efficacité.

6.6.1 Création de cas de test

La création des cas de test est réalisée en trois étapes majeures, incluant deux réunions. Voici la procédure à suivre :

1- Séance collective de créativité (*Brainstorming*) : C'est une technique utilisée dans les réunions pour encourager la contribution simultanée des idées de tous les membres du groupe. Tous les membres de l'équipe qui ont fait du code *Walkthrough* doivent participer à cette séance. Avant de se présenter à cette séance, chaque membre doit préparer les documents produits à l'étape 12 de la procédure du code *Walkthrough* (voir paragraphe 6.3). Cette séance collective de créativité (Brainstorming) est nécessaire pour produire le plus de cas de test qui répondront à nos besoins tout en se basant sur les résultats trouvés lors des étapes précédentes. Au cours de cette séance, tous les membres de l'équipe doivent prendre des notes qui vont les aider lors de l'étape suivante. À la fin de cette réunion, un membre de l'équipe se chargera de produire un document contenant les cas de test créés lors de cette séance. Il devra les distribuer à tous les membres.

2- Analyse : Après avoir terminé la séance collective de créativité, chacun des membres de l'équipe doit faire une analyse complète des résultats obtenus lors de l'étape précédente (*brainstorming*) ainsi que ceux obtenus lors de la phase du *Walkthrough*. Après cette analyse, un document contenant les cas de test, les questions et les remarques doit être généré individuellement.

3- Discussion et démonstration : Une réunion de tous les membres de l'équipe est convoqué. Dans cette réunion chacun présentera les cas de test produits afin d'obtenir les commentaires des autres. De plus, l'utilité de chaque cas de test devra être démontrée et approuvée. Un membre de l'équipe se chargera de produire un document contenant les cas de test finaux.

Une fois les cas de test créés et bien documentés on passe à la phase suivante. C'est-à-dire, la phase d'écriture des procédures de test et du plan de test.

6.6.2 Distribution temporelle des activités

La création des cas de test se fait à travers les trois phases illustrées à la figure 6.1 : phase de *Brainstorming*, phase d'analyse et phase de discussion et de démonstration. Cette section présente sous forme graphique les résultats des données recueillies lors de l'exécution du projet. Ce projet implique 5 personnes durant 4 mois. Le but de la figure 6.2 est de présenter le pourcentage de temps qu'occupe chaque phase pour créer les cas de test.

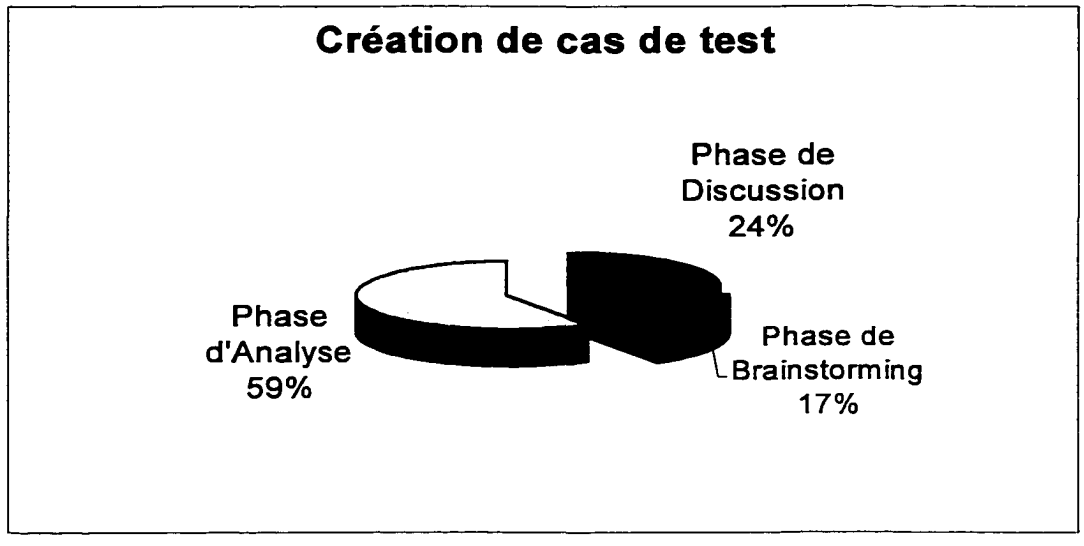


Figure 6.2 Pourcentages temporels des trois phases pendant la création des cas de test

Parmi ces trois phases (*Brainstorming*, analyse et discussion et démonstration), il y a deux réunions : réunion pour du *Brainstorming* et réunion pour discussion et démonstration. Comme on l'a vu dans la section 6.5.1, dans chaque réunion il y a plusieurs activités. Dans la figure 6.3 nous montrons la distribution temporelle de ces activités en pourcentage.

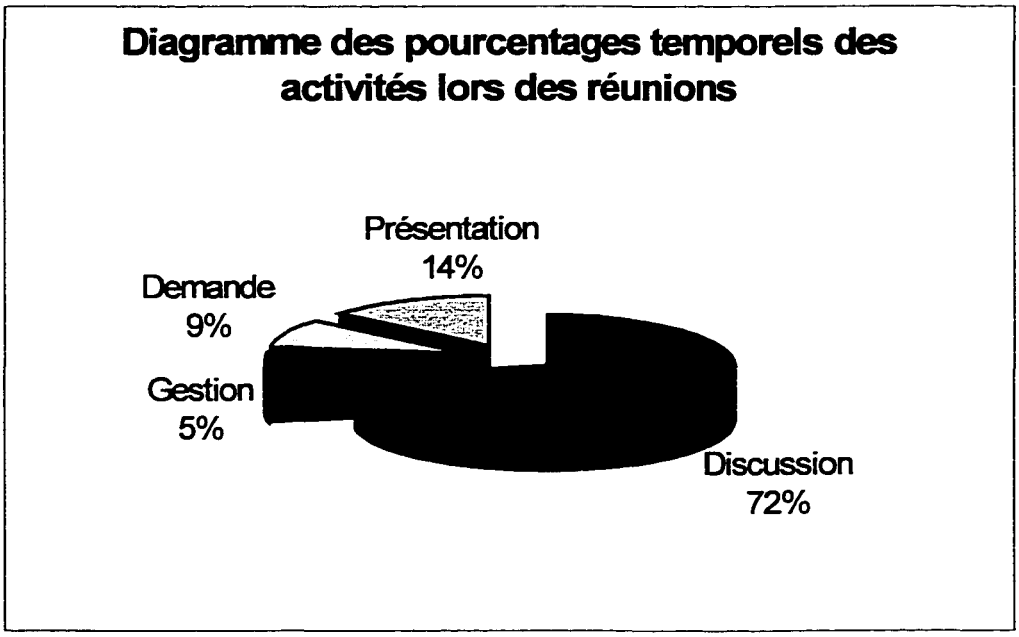


Figure 6.3 Pourcentages temporels des activités lors des réunions dans les phases de *Brainstorming* et de discussion et de démonstration

On remarque que l'activité de discussion occupe 72% du temps des réunions. C'est pourquoi on l'a divisé en 7 sous-activités (voir section 6.5.1). Pour notre projet, voici les pourcentages temporels de chacune de ces sous-activités.

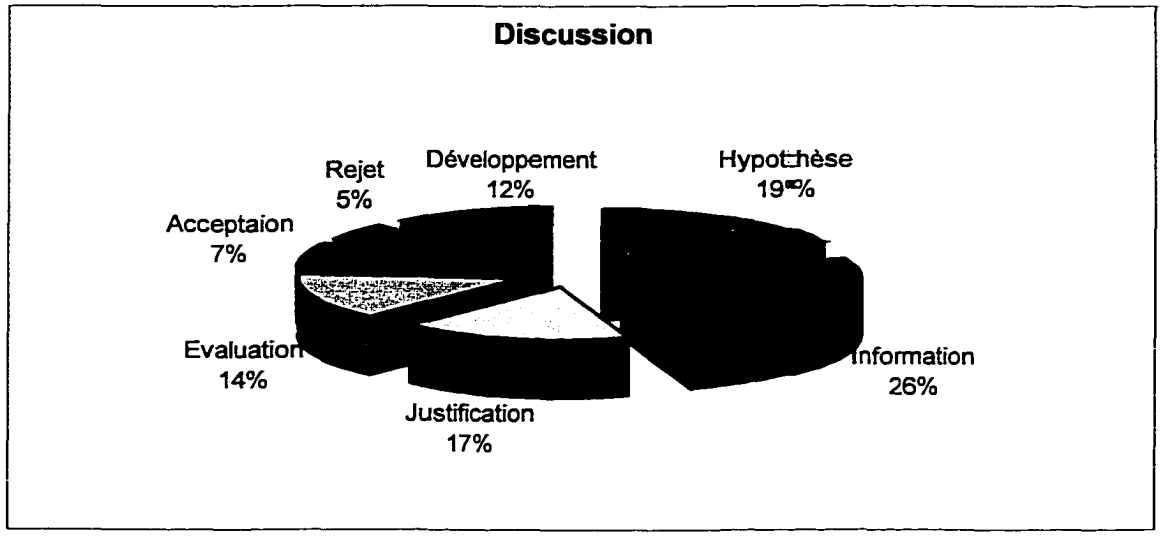


Figure 6.4 Pourcentages temporels des sous-activités de l'activité de discussion

6.7 Plan de test

Une fois toutes les phases précédentes réalisées, on passe à la phase de l'élaboration d'un document intitulé «plan de test». Un modèle présentant le format d'un tel plan est disponible en annexe de ce mémoire.

En se basant sur les normes du standard IEEE 829-1983 [10], et de notre expérience dans l'industrie, un plan de test doit contenir les sections suivantes :

1- Identification du plan de test

Cette section spécifie l'identification unique du plan de test. L'identification pourrait être tout simplement un titre ou toute autre nomenclature formelle assignée par la gestion de la configuration. Aussi elle pourrait être une page titre contenant : le numéro du document, le volume, la version, le titre, le(s) nom(s) du système(s) et sous-système(s) les abréviations, le nom de la compagnie qui a préparé le document, le nom de la compagnie pour laquelle le document est préparé et tout autre élément d'identification pertinent.

2- Table de matière

Cette table contient le titre, le numéro de la page, les figures, les tables, les appendices et les références.

3- Introduction

L'introduction doit inclure les paragraphes suivants :

a. Le but du projet
b. Une description du logiciel sous test. Ce paragraphe doit donner un aperçu du logiciel, ses opérations, maintenance, histoire, code et toute autre information pertinente.
c. Aperçu du document. Ce paragraphe décrit le document en général.

De plus, une liste de références pour le projet pourrait être incluse dans le plan de test. Selon le standard de l'IEEE, les références aux documents suivants, s'ils existent, doivent être mentionnées :

a. Autorisation du projet ;
b. Plan du projet ;
c. Plan de l'assurance qualité ;
d. Plan de la gestion de la configuration ;
e. Politiques pertinentes de la compagnie et du client ;
f. Standards pertinents de la compagnie, du client, et de l'industrie.

4- Relations avec d'autres documents.

Cette section cite et décrit les relations avec d'autres documents.

5- Les références.

Ce paragraphe doit mentionner les détails de tous les documents auxquels on fait référence dans ce document.

6- Les documents applicables

Ce paragraphe doit mentionner les détails de tous les documents applicables à ce projet et qui en forment une partie.

7- Le site et la date

Ce paragraphe doit identifier le(s) site(s) où les tests auront lieu et la date du début des tests.

8- Le personnel de test

Ce paragraphe doit identifier le personnel affecté à la réalisation des tests. La fonction de chaque personne ainsi que l'identification de son organisation doivent être mentionnées.

9- Les organisations participantes

Ce paragraphe doit identifier les organisations participantes aux tests, leurs rôles et leurs responsabilités.

10- Restrictions

Ce paragraphe doit identifier les normes de sécurité et les restrictions applicables à ce document.

11- Les éléments de test

Il faut spécifier les éléments de test nécessaires pour monter le banc de test. Les versions du matériel et du logiciel doivent être spécifiées à ce niveau. De plus, il faut mentionner les documentations utilisées dans le projet (manuel, guide d'installation, etc.)

12- Les unités à tester :

Dans cette section il faut identifier toutes les unités ou modules à tester.

13- Les fonctions à tester :

Dans cette section il faut identifier toutes les fonctions et les combinaisons de fonctions à tester.

14- Les niveaux de test :

Dans cette section il faut décrire tous les niveaux auxquels les tests vont être faits.

15- La progression de test

Ici on doit décrire la planification de la séquence de test.

16- La sauvegarde des résultats

Cette section doit décrire le moyen utilisé pour sauvegarder certains résultats de test.

17- Approche :

Ici on doit décrire l'approche de test utilisé et les outils. La méthodologie de test, l'efficacité de l'outil et la sévérité des tests doivent être clairs.

18- Critère du succès ou d'échec

Dans cette section le testeur doit spécifier les critères qui déterminent le succès ou l'échec d'un test. Le testeur peut aussi déterminer le niveau de criticalité de son échec et de son succès suivant des métriques qu'il aura à fixer. Prenons l'exemple d'un système capable d'identifier que le 30 février 2000 n'est pas une date valide et de la convertir en 1er mars 2000 sans en avertir l'usager. Ce système n'a pas passé le test de l'an 2000, mais il n'a pas échoué non plus. Le testeur peut définir dans ce cas une nouvelle catégorie qui pourrait être appelée «succès limité ».

19- Livrables de test

Cette section définit la liste des documents qu'on doit envoyer au client après avoir complété les tests. Voici, quelques-uns des livrables tels que décrits par l'IEEE :

a. Plan de test ;
b. Spécifications de conception de test ;
c. Spécifications de cas de test ;
d. Spécifications de procédure de test ;
e. Logs de test ;
f. Rapports incidents ;
g. Rapports sommaires ;
h. Outils de test ;

i. Données d'entrées et de sorties du test.

Il faut rajouter à ces documents, un document certifiant le bon fonctionnement du logiciel selon l'exigence du client. Par exemple, une certification du passage à l'an 2000 du logiciel.

20- Les besoins de test

Tout le support nécessaire en logiciel et en matériel doit être mentionné ici. L'environnement, l'endroit de test, les caractéristiques des lieux et tout ce qui est nécessaire pour tester le logiciel doit être décrit.

21- Responsabilité

Dans cette section il faut identifier le personnel ayant contribué à ce projet et décrire sa responsabilité. Toute personne participant doit être identifiée et ses tâches doivent être décrites.

22- Formation du personnel

Les besoins en formation du groupe et du personnel assigné pour ce projet doivent être spécifiés dans cette section. De plus, les noms des personnes contacts qui doivent être présentes lors de l'exécution des tests doivent être mentionnés.

23- Échéancier

Les dates de début des tests, fin de test, durée de chaque phase et tout autres événements doivent être spécifiés.

24- Risques et contingences

Dans cette section il faut mentionner les choses qui pourraient mal tourner. Par exemple, dans un test de l'an 2000, il faut spécifier que les changements de date exécutés lors des tests pourraient provoquer l'expiration de certaines licences. De même, il faut

aussi spécifier les choses qui pourront mal tourner de façon catastrophique. Par exemple, un avancement de date, pour une date future, mais le logiciel ne peut plus revenir à la date antérieure après le changement. Dans un tel cas, le système est coincé à une mauvaise date et les opérations sont arrêtées. Un plan de contingence devrait être prévu pour de telles situations.

25- Approbation

Dans cette section il faut mentionner les noms des personnes qui ont approuvées le plan.

6.8 Analyse des résultats

La méthode de test que nous avons introduit au cours de ce chapitre offre la possibilité de voir le code pour ensuite bâtir des cas de test. Ceci permet d'exercer des tests plus efficaces et plus exhaustifs. Un simple test de boîte noire ne permet pas de couvrir tous les cas possibles car il est basé seulement sur les fonctionnalités externes du système. Pour bien illustrer l'efficacité de notre méthode nous allons donner l'exemple suivant.

Nous avons fait des tests de boîte noire pour les anomalies relatives à la finitude des grandeurs (voir chapitre 5) pour le logiciel sous test. Toutes les anomalies trouvées étaient réglées et tous les cas de tests ont été refaits et ont tous passé sans aucun échec. Alors, selon ces tests de boîte noire on pourrait certifier que le logiciel ne contient pas d'anomalies relatives à la finitude des grandeurs. À la page suivante on trouve un exemple d'un cas :

Tableau 6.1

Exemple d'un cas de test pour vérifier le problème du overflow des systèmes UNIX

<table>
<tr><td colspan="4">Identification du cas de test : YT CT-32</td></tr>
<tr><td colspan="4">But : Valider que le système fonctionnera sans anomalies à la date du overflow des systèmes UNIX, soit le 18 janvier 2038</td></tr>
<tr><td colspan="4">Pre-conditions : YT PY-3</td></tr>
<tr><td colspan="4">Entrées : La date du 17 janvier 2038 23:59</td></tr>
<tr><td colspan="4">Procédure : - Avancer la date du système au 17 janvier 2038 23:59
- Laisser le système rouler pendant une minute</td></tr>
<tr><td colspan="4">Post-Conditions : YT TY-5</td></tr>
<tr><td colspan="4">Sorties prévues : La date du système est le 18 Janvier 2038</td></tr>
<tr><td colspan="4">Sortie : la date est le 18 janvier 2038 00:17:00 EST</td></tr>
<tr><td colspan="4">Dépendances : YT CT -52, YT CT -89, YT CT –79</td></tr>
<tr><td colspan="4">Traçage : TZ RM-54</td></tr>
<tr><td>Date</td><td>Version</td><td>Résultat</td><td>Exécuté par</td></tr>
<tr><td>12/06/99</td><td>1.2.1 r.12</td><td>Succès</td><td>H.Z.</td></tr>
</table>

Nous avons ensuite appliqué notre méthode afin de tester le même logiciel pour les anomalies relatives à la finitude des grandeurs. Grâce à notre méthode, qui est basée sur l'inspection des parties ciblées du code, nous avons pu trouver certaines routines dont le comportement n'est pas évident. En effet, ces routines s'exécutent toutes seules sans aucune intervention de la part de l'usager et de l'administrateur. Et ce, à toutes les 24 heures et d'autres à toutes les 48 heures. Alors, nous avons décidé d'inclure un cas de test qui demande de laisser le système s'exécuter pendant 48 heures à une date de transition critique concernant les anomalies de la finitude de grandeurs.

Tableau 6.2

Exemple d'un cas de test pour vérifier le problème du overflow des systèmes UNIX en appliquant notre méthode

Identification du cas de test : YT CT-73			
But : Valider que le système fonctionnera sans anomalies à la date du overflow des systèmes UNIX, soit le 18 janvier 2038, tout en laissant les routines de 48 heures s'exécuter			
Pre-conditions : YT PY-3			
Entrées : La date du 17 janvier 2038 23:59			
Procédure : - Avancer la date du système au 17 janvier 2038 23:59 - Laisser le système rouler pendant 49 heures - Vérifier les informations dans les bases de données qui sont stockés en faisant des calculs des dates ou de nombres de jours écoulés.			
Post-Conditions : YT TY-5			
Sorties prévues : - La date du système est le 18 Janvier 2038 - Les informations dans la base de données sont correctes.			
Sortie : - la date est le 18 janvier 2038 00:17:00 EST **- La date de certaines informations dans les bases de données est le 14 juin 2100** **- Certaines informations dans la base de données sont complètement erronées**			
Dépendances : YT CT -52, YT CT –54, YT CT -79, YT CT –89			
Traçage : TZ RM 57			
Date 07/10/99	Version 1.2.1 r.12	Résultat Echec	Exécuté par H.Z.

Comme on a pu le constater, en laissant le système s'exécuter pendant 48 heures consécutives nous avons déclenché l'exécution autonome des routines. Ces routines effectuent des requêtes d'informations stockées dans des bases de données suivant leurs dates d'enregistrement. Elles réalisent ensuite certains calculs et comparaisons en fonction de la date de ces informations. Les résultats sont finalement stockés dans une base de donnés.

Comme le cas de test concernant le problème de débordement des systèmes UNIX a passé sans problème nous pouvons donc conclure que le système ne devrait pas présenter d'anomalies à cette date là. La question qui se pose maintenant est : « pourquoi les anomalies se manifestent-elles lors de l'exécution du logiciel ? ».

Quand on a corrigé le code pour qu'il soit compatible avec les problèmes de la finitude des grandeurs, nous n'avons pas corrigé le code de ces routines qui n'acceptaient pas de dates à 4 chiffres. En faisant un test fonctionnel, nous ne pouvions pas nous rendre compte de ces routines qui s'exécutent sans intervention de la part de l'usager ou de l'administrateur. Notre méthode est basée sur l'inspection des parties du code en relation avec les anomalies qu'on cherche. Alors, les cas de test ne vérifient pas seulement les fonctionnalités à court terme du système, mais aussi toutes autres routines internes ou parties du code en relation avec les anomalies. L'exemple que nous venons de voir montre que sans l'utilisation de notre méthode, le logiciel aurait été certifié comme étant sans anomalies. L'application de notre méthode a relevé la présence des anomalies dans le logiciel ce qui a empêché l'interruption des opérations du système. Alors, nous pouvons voir avec cet exemple l'efficacité de notre méthode et les avantages qu'elle apporte à l'industrie du logiciel.

6.9 Avantages de notre méthode

Avec la méthode de boîte noire brute, la raison de l'échec d'un test n'est pas évidente. Tandis qu'avec notre méthode, nous pouvons le savoir à l'avance ou le trouver très facilement car les cas de test ont été bâtis en se basant sur les parties de code inspecté.

De plus, avec la méthode de boîte noire brute, les résultats sont surestimés tandis qu'avec notre méthode nous pourrons facilement évaluer nos résultats puisqu'on connaît l'origine de ces résultats. Un autre facteur très important de notre méthode est de rendre les cas de test plus efficaces et permettre plus de couverture. Parce qu'en inspectant le

code on trouve plus de cas de test, on se concentre sur les parties du code qui nous intéressent.

Quant à la méthode de boîte blanche bête, elle exige des cas de test qui assurent que tous les chemins indépendants, toutes les décisions logiques et toutes les boucles soient exercés. Ceci est trop coûteux en terme d'argent et de temps dans le cas de grands logiciels. En plus, dans le contexte des anomalies traitées dans ce mémoire la période de temps allouée pour le projet est relativement courte. Un autre facteur limitatif est que l'établissement qui exécute les tests est totalement indépendant de l'établissement développeur du logiciel. Notre méthode est taillée pour les compagnies qui exécutent des tests et permet la réduction du temps d'exécution comparée à la méthode de boîte blanche bête.

Voici d'autres avantages de notre méthode :

1- On peut reproduire les tests ;
2- L'environnement du logiciel est aussi testé ;
3- L'effort investi pourrait être utilisé plusieurs fois ;
4- On n'a pas besoin d'inspecter tout le code, mais seulement les parties qui concerne notre type de test, ce qui réduit énormément le temps d'inspection ;
5- Contrairement à la méthode de boîte noire conventionnelle. Les cas de test sont bâtis en fonction du code. Ce qui les rend plus efficaces.

6.10 Conclusion

L'objectif principal de ce chapitre était de présenter notre méthode de test. Nous avons commencé par définir la revue technique formelle, l'inspection et le *walkthrough*. Ensuite nous avons décrit le processus complet de notre méthode de test. Nous avons défini et expliqué les activités et les livrables à chaque étape et nous avons spécifié le contenu des documents générés aux différentes étapes. Nous avons aussi donné les statistiques des activités qui se déroulent lors des réunions techniques. Nous avons

ensuite terminé le chapitre en mettant l'accent sur l'importance de notre méthode et les solutions qu'elle apporte aux problèmes de tests dans l'industrie du logiciel.

CONCLUSION

Nous avons étudié dans ce travail différentes méthodes permettant de réaliser des tests logiciels. L'objectif premier de cette étude consistait à obtenir une méthode permettant de trouver des cas de test efficaces pour des grand logiciels (plus que 2.5 millions lignes de code) en un temps raisonnable et de résoudre les problèmes associés au test en groupe.

Au début de ce travail nous avons introduit le lecteur au test de logiciels et aux différentes méthodes et techniques utilisées dans ce domaine. Ensuite nous avons consacré un chapitre complet pour présenter le test qui concerne notre travail, soit le test fonctionnel.

Nous avons abordé les problèmes rencontrés tels l'exécution des fonctions complexes, la communication avec le matériel, l'utilisation de plus d'un langage de programmation. Ensuite, nous avons discuté de l'impossibilité d'un test complet et l'implication des problèmes de type NP et d'explosion combinatoire.

Par la suite, nous avons présenté les types d'anomalie que nous cherchions à débusquer soit les anomalies logiciels relatives aux problèmes de la finitude des grandeurs. Nous avons exposé ce problème en détail, montré les systèmes qu'il peut toucher et les domaines où il se pose. Ensuite nous avons dressé et analysé les différentes causes qui ont provoqué ce problème. Nous avons consacré une partie du chapitre cinq pour discuter de la partie statique de notre méthode de test. C'est-à-dire, la partie qui consiste en l'inspection faite en groupe assisté d'outil d'automatisation. En plus, dans ce chapitre, nous avons décrit la démarche à suivre pour créer un dictionnaire contextuel. Ce

dernier contient tous les mots clés concernant le problème relatif aux anomalies cherchées dans ce mémoire. Nous avons montré le besoin d'une méthode qui nous aidera à réduire le temps nécessaire pour créer les cas de test et de les rendre plus efficace.

Le chapitre six continue dans ce cheminement et explique plus en détail notre méthode de test. Les deux méthodes les plus répandues sont la méthode de boîte blanche et la méthode de boîte noire. La première est presque impossible dans le contexte des types d'anomalies abordés dans notre mémoire à cause du temps énorme qu'elle nécessite pour exécuter tous les tests. La deuxième, boîte noire, n'assure pas une couverture complète des tests et ne permet pas de savoir l'origine de l'anomalie. La méthode que nous avons présenté dans ce mémoire repose sur la méthode de boîte noire et la méthode de revue de code intégrées dans un processus bien défini. Notre méthode consiste en un mélange de deux méthodes. L'une statique et l'autre dynamique dans un même processus. Ceci est réalisé en faisant de la revue de code dans un environnement de groupe assisté par ordinateur et des tests fonctionnels dans une seule méthode.

Le processus complet et détaillé de notre méthode a été identifié au cours de ce mémoire. Nous avons fourni un organigramme de notre méthode avec l'explication des activités et les livrables de chaque étape afin de faciliter son utilisation. De plus nous avons spécifié en détail le contenu d'un plan de test qui pourrait servir comme modèle standard.

Une analyse des résultats a montré que notre méthode a relevé plus de cas de test que la méthode boîte noire pour un même logiciel. L'exemple que nous avons vu au cours de ce mémoire a montré que l'application de notre méthode a permis de trouver des anomalies dans un logiciel qui a été testé par la méthode de boîte noire et a été certifié ne pas contenir des anomalies. Alors, nous pouvons constater les coûts et les problèmes que notre méthode a pu épargner en trouvant plus d'anomalies que la méthode de boîte noire.

Ce mémoire est un des premiers travaux en français sur les tests de logiciels. Elle fournit à l'industrie de logiciel une méthode et un outil lui permettant de mieux gérer le processus de test de grands logiciels complexes. De créer des cas de test plus efficaces et reproductibles et surtout de réduire le temps nécessaire pour élaborer les cas, le plan et la procédure de test. De plus, et contrairement à la méthode de boîte noire, nous serons en mesure d'expliquer la raison de l'anomalie trouvée en utilisant notre méthode proposées. Toutes les étapes, les difficultés qui peuvent surgir au cours du processus et les livrables à chaque étape ont été présentés et détaillés dans ce mémoire.

Recommandations

Au cours de ce mémoire nous avons développé un dictionnaire contextuel pour nous aider à trouver les anomalies reliées au problème de la finitude de grandeur. Un travail futur sera de développer un dictionnaire contextuel pour le problème de la conversion des systèmes informatiques vers l'Euro, un problème qui préoccupe la majorité des établissements informatique. Un autre travail sera de bâtir un dictionnaire contextuel et appliquer notre méthode pour trouver les anomalies reliées à la modification d'un programme informatique pour le faire exécuter d'une façon différente, sous certaines circonstances, ce problème est connu sous le nom de Bombe Logique [34].

Nous suggérons l'application de notre méthode et la méthode de boîte blanche ou boîte noire sur un même logiciel et la collection des métriques pour ensuite faire de comparaison afin d'établir la fiabilité et l'efficacité de notre méthode.

Bibliographie

[1] ANDREWS, E. L. (1991). Computer maker says tiny software flaw caused phone disruptions. N.Y. Times, 10 juillet 91.

[2] Jézéquel, J. M., Meyer, B. (1997). Design by Contract: The Lessons of Ariane. IEEE Computer, 70(2), pp. 129-130.

[3] Colonna, J. F. (1999). Doit-on craindre l'an 2000 ? ou les problèmes de gestion de dates dans les ordinateurs. http://www.lactamme.polytechnique.fr/Mosaic/descripteurs /An2000.01.Fra.html

[4] Myers, G.J. (1979). The Art of Software Testing. New york : Wiley-Interscience.

[5] IEEE Std 729-1983. (1983). IEEE Computer Society: *IEEE Standard Glossary of Software Engineering Terminology,* The Institute of Electrical and Electronic Engineers.

[6] IEEE Standards Board - 1059. (1993). IEEE Guide for Software Verification and Validation Plans. New York : The Institute of Electrical and Electronic Engineers.

[7] Beizer, B. (1995). Black Box Testing. New York : John Wiley & Sons, Inc.

[8] Hetzel, W.C. (1988). The Complete Guide to Software Testing. New York : John Wiley & Sons, Inc.

[9] Rook, P. (1990). Software Reliability Handbook. London : Elsevier applied science.

[10] IEEE Standard 829. (1983). Standard for software test documentation. New York IEEE Press.

[11] Jorgensen, P.C. (1995). Software Testing. Boca Raton : CRC Press.

[12] Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary : A Compilation of IEEE Standard Computer Glossaries.* New York, NY: 1990.

[13] IEEE Standards Board - IEEE 610.12. (1991). IEEE Standard Glossary of Software Engineering Terminology (corrected edition). New York: American National Standards Institute.

[14] Maquire, S. (1993). Writing solid code. Redmond : Microsoft Press.

[15] Thaller, G. (1993). Qualitätsoptimierung der Software-Entwicklung - Das Capability Maturity Model (CMM). Vieweg : Braunschweig.

[16] Rozenberg, M. (1998). Test logiciel. Paris : Eyrolles.

[17] Pressman, R.S. (1997). Software Engineering : a practitioner's approach. McGraw Hill.

[18] Kaner C., Falk, J., Nguyen, H. Q. (1993). Testing Computer Software (2e éd.). Boston: International Thomson Computer Press.

[19] Beizer, B. (1990). Software Testing Techniques (2e éd.). New York : Van Nostrand Reinhold.

[20] Elmendorf, W. R. (1973). Cause effect graphs in functional testing. New York : IBM systems developpment division.

[21] Perry, W. (1995). Effective methods for software testing. New York : John Willey & Sons.

[22] Mcconnel, S. (1993). Code Complete : A Practical Handbook of Software Construction. Redmond : Microsoft Press.

[23] Yourdon, E. (1982). Managing the system life cycle. New York : Yourdon Press.

[24] Manna, Z., Waldinger, R. (1978). The logic computer programming. IEEE Transactions on software engineering SE-4, 24(26), pp. 199-229.

[25] Garey, M.R., Johnson, D.S. (1979). Computers and intractability: A guide to the theory of NP-completeness. San Francisco: W.H. Freeman and Company.

[26] Hetzel, B. (1988). The complete guide to software testing (2e éd.). New York : John Willey & Sons.

[27] Kaner, C. (1997) The Impossibility of Complete Testing. Software QA magazine. http://www.kaner.com/imposs.htm

[28] FAGAN, M.E. (1976). Inspection Software Design and Code. IBM Systems Journal, 15(3), pp. 182-211.

[29] Johnson, P..M. (1996). Reengineering Inspection: The Future of Formal Technical Review. Honolulu : University of Hawaii.

[30] BRYCZYNSKI, B., MEESON, R.N., WHEE-LER, D.A. (1994). Software Inspection: Eliminating Software Defects. Proceedings of the sixth annual software technology conference, Alexendria.

[31] IEEE Std 1028. (1993). IEEE Computer Society: *IEEE Standards collection for Software Engineering*, The Institute of Electronical and Electronics Engineers.

[32] PORTER A., SIY, H., MOCKUS, A., VOTTA, L.(1997). Understanding the Sources of Variation in Software Inspections. Technical Report at University of Maryland (UMCP-CSD:CS-TR-3762).

[33] d'Astous, P., Robillard, P. (1997). La mesure des activités collaboratives retrouvées lors d'une réunion de révision technique du processus de génie logiciel. École Polytechnique de Montréal.

[34] Overview of the risks, part 2, David J. Stang, Ph.D www.sevenlocks.com/security/NSS01BAnOverviewoftheRiskspart2of2.htm

[35] Rakitin, S. R. (1997). Software Verification and Validation. Boston : Artech House.

[36] Sobey, A. J. University of south Australia. (1997). Black box testing, http://louisa.levels.unisa.edu.au/se1/testing-notes/test01_5.htm

GLOSSAIRE

Anomalie

Incompatibilité entre le fonctionnement de l'application et les exigences fonctionnelles et techniques.

Cas de test

Les objectifs détaillés, les conditions, les données (entrées et sorties), les procédures et les résultats prévus pour effectuer un test.

Débogage

Localisation et élimination des erreurs dans un programme. Des routines de diagnostic sont exécutées pour localiser l'erreur.

Inspection de Fagan

Une technique de test [détection des erreurs] manuelle où le programmeur lit le code source ligne par ligne à un groupe qui pose des questions analysant la logique du programme et sa conformité aux normes de codage.

Plan de test

L'information détaillée pour la conception de test, des cas de test, et la réalisation de test. Le plan d'essai inclut les objectifs, l'échéancier, les ressources exigées, les dépendances, les responsabilités, les livrables, etc.

Procédure

Description écrite de la marche à suivre pour l'accomplissement d'une tâche donnée.

RTC (Real Time Clock)

C'est une horloge au niveau de matériel qui maintient l'heure et la date même lorsque la machine est arrêtée.

Walkthrough du code

Une technique de test manuel (détection des erreurs) où la logique (structure) de programme (code source) est tracée manuellement et analysée.

ANNEXE A

Plan de Test
(Modèle)

Organisation de ce modèle de plan de test :

Chaque élément ou paragraphe dans ce modèle est identifié par un numéro unique. Ces numéros vont par ordre croissant en allant de «1. Nom de l'organisation » jusqu'à «34. Approbation ».

<table>
<tr><td colspan="3">1. Nom de l'organisation
[Insérer le nom de l'organisation ici]</td></tr>
<tr><td colspan="3">2. Titre
Plan de test</td></tr>
<tr><td colspan="2">3. Nom du Projet
[Insérer le nom du projet ici]</td><td>4. Numéro d'identification
[Insérer le numéro d'identification ici]</td></tr>
<tr><td>5. Nom du module
[Insérer le nom du module ici]</td><td colspan="2">6. Organisations Impliquées et rôles
[Insérer les noms des organisations impliquées et leurs rôles ici]</td></tr>
<tr><td colspan="3">7. But
[Insérer le but ici]</td></tr>
<tr><td colspan="3">8. Numéro d'identification des documents applicables
[Insérer les numéros ici]</td></tr>
<tr><td>9. Classification
[Insérer la classification ici]</td><td colspan="2">10. Date
[Insérer la date ici]</td></tr>
</table>

11. Table de matière

12 - Introduction

L'introduction doit contenir les paragraphes suivants :

a. Le but du projet;
b. Description du logiciel sous test : Ce paragraphe doit donner un aperçu du logiciel, ses opérations, maintenance, histoire, code, et toute autre information pertinente;
c. Aperçu du document : Ce paragraphe décrit le document en général;

De plus, une liste de références pour le projet pourrait être incluse dans le plan de test. Selon le standard de l'IEEE, les références aux documents suivants, s'ils existent, doivent être mentionnées :

a. Autorisation du projet;
b. Plan du projet;
c. Plan de l'assurance qualité;
d. Plan de la gestion de la configuration;
e. Politiques pertinentes de la compagnie et du client;
f. Standards pertinents de la compagnie, du client et de l'industrie.

13- Relations avec d'autres documents

Cette section cite et décrit les relations avec d'autres documents.

14- Les références

Ce paragraphe doit mentionner les détails de tous les documents auxquels on fait référence dans ce document.

15- Les documents applicables

Ce paragraphe doit mentionner les détails de tous les documents applicables à ce projet et qui en forment une partie.

16- Le site et la date

Ce paragraphe doit identifier le(s) site(s) où les tests auront lieu et la date du début des tests.

17- Le personnel de test

Ce paragraphe doit identifier le personnel affecté à la réalisation des tests. La fonction de chaque personne ainsi que l'identification de son organisation doivent être mentionnées.

18- Les organisations participantes

Ce paragraphe doit identifier les organisations participantes aux tests, leurs rôles et leurs responsabilités.

19- Restrictions

Ce paragraphe doit identifier les normes de sécurité et les restrictions applicables à ce document.

20- Les éléments de test

Il faut spécifier les éléments de test nécessaires pour monter le banc de test. Les versions du matériel et du logiciel doivent être spécifiées à ce niveau. De plus, il faut mentionner les documentations utilisées dans le projet (manuel, guide d'installation, etc.)

21- Les unités à tester

Dans cette section il faut identifier toutes les unités ou modules à tester.

22- Les fonctions à tester

Dans cette section il faut identifier toutes les fonctions et les combinaisons de fonctions à tester.

23- Les niveaux de test

Dans cette section il faut décrire tous les niveaux auxquels les tests vont être faits.

24- La progression de test

Ici on doit décrire comment on a planifié la séquence de test.

25- La sauvegarde des résultats

Cette section doit décrire le moyen utilisé pour sauvegarder certains résultats de test.

26- Approche

Ici on doit décrire l'approche de test utilisé et les outils. La méthodologie de test, l'efficacité de l'outil et la sévérité des tests doivent être clairs.

27- Critère du succès ou d'échec

Dans cette section le testeur doit spécifier les critères qui déterminent le succès ou l'échec d'un test. Le testeur peut aussi déterminer le niveau de criticalité de son échec et de son succès suivant des métriques qu'il aura à fixer. Prenons l'exemple d'un système capable d'identifier que le 30 février 2000 n'est pas une date valide et de la convertir en 1er mars 2000 sans en avertir l'usager. Ce système n'a pas passé le test de l'an 2000, mais il n'a pas échoué non plus. Le testeur peut définir dans ce cas une nouvelle catégorie qui pourrait être appelée «succès limité ».

28- Livrables de test

Cette section définit la liste des documents qu'on doit envoyer au client après avoir complété les tests. Voici, quelques-uns des livrables tels que décrits par l'IEEE :

a. Plan de test;
b. Spécifications de conception de test;
c. Spécifications de cas de test;

d. Spécifications de procédure de test;
e. Logs de test;
f. Rapports incidents;
g. Rapports sommaires;
h. Outils de test;
i. Données d'entrées et de sorties du test.

Il faut rajouter à ces documents, un document certifiant le bon fonctionnement du logiciel selon l'exigence du client. Par exemple, une certification du passage à l'an 2000 du logiciel.

29- Les besoins de test

Tout le support nécessaire en logiciel et en matériel doit être mentionné ici. L'environnement, l'endroit de test, les caractéristiques des lieux et tout ce qui est nécessaire pour tester le logiciel doit être décrit.

.

30- Responsabilité

Dans cette section il faut identifier le personnel ayant contribué à ce projet et décrire sa responsabilité. Toute personne participant doit être identifiée et ses tâches doivent être décrites.

31- Formation du personnel

Les besoins en formation du groupe et du personnel assigné pour ce projet doivent être spécifiés dans cette section. De plus, les noms des personnes contacts qui doivent être présentes lors de l'exécution des tests doivent être mentionnés.

32- Échéancier

Les dates de début des tests, fin de test, durée de chaque phase et tout autres événements doivent être spécifiés.

33- Risques et contingences

Dans cette section il faut mentionner les choses qui pourraient mal tourner. Par exemple, dans un test de l'an 2000, il faut spécifier que les changements de date exécutés lors des tests pourraient provoquer l'expiration de certaines licences. De même, il faut aussi spécifier les choses qui pourront tourner mal de façon catastrophique. Par exemple, un avancement de date, pour une date future, mais le logiciel ne peut plus revenir à la date antérieure après le changement. Dans un tel cas, le système est coincé à une mauvaise date et les opérations sont arrêtées. Un plan de contingence devrait être prévu pour de telles situations.

34- Approbation

Dans cette section il faut mentionner les noms des personnes qui ont approuvées le plan.