UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

JEAN-FRANÇOIS MICHAUD

B. ING., D.E.S.S.

MÉTHODE GÉNÉRIQUE DE GESTION DES TESTS D'UNITÉS LOGICIEL

DÉCEMBRE 2005

CE MÉMOIRE A ÉTÉ RÉALISÉ

À L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

DANS LE CADRE DU PROGRAMME

DE MAÎTRISE EN INFORMATIQUE

DE L'UNIVERSITÉ DU QUÉBEC À MONTRÉAL

OFFERT PAR EXTENSION À L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

Mise en garde/Advice

Afin de rendre accessible au plus grand nombre le résultat des travaux de recherche menés par ses étudiants gradués et dans l'esprit des règles qui régissent le dépôt et la diffusion des mémoires et thèses produits dans cette Institution, **l'Université du Québec à Chicoutimi (UQAC)** est fière de rendre accessible une version complète et gratuite de cette œuvre.

Motivated by a desire to make the results of its graduate students' research accessible to all, and in accordance with the rules governing the acceptation and diffusion of dissertations and theses in this Institution, the **Université du Québec à Chicoutimi (UQAC)** is proud to make a complete version of this work available at no cost to the reader.

L'auteur conserve néanmoins la propriété du droit d'auteur qui protège ce mémoire ou cette thèse. Ni le mémoire ou la thèse ni des extraits substantiels de ceux-ci ne peuvent être imprimés ou autrement reproduits sans son autorisation.

The author retains ownership of the copyright of this dissertation or thesis. Neither the dissertation or thesis, nor substantial extracts from it, may be printed or otherwise reproduced without the author's permission.

RÉSUMÉ

L'objectif de ce mémoire de maîtrise en informatique est de présenter une méthode générique de gestion des tests d'unités logiciel utilisable par la très petite entreprise (2 à 10 programmeurs). Cette méthode répond aux contraintes que rencontrent ces entreprises : faible budget et développement/maintenance en plusieurs langages informatiques ou environnements. De même, cette méthode est simple à implémenter et à enseigner aux nouveaux programmeurs. Pour ce faire, ce document regroupe les informations pertinentes menant à la compréhension de la place qu'occupent les tests dans les différentes méthodes de développement de logiciel. De même, il montre un échantillonnage de plusieurs méthodes usuelles de tests qui peuvent être utiles aux programmeurs.

La méthode de gestion de tests développée dans ce document est basée sur une utilisation des logiciels de tests et sur les tests de régression. De plus, cette méthode est générique car elle s'adresse à tous les langages informatiques de par l'utilisation des logiciels de tests et dû au fait que l'implémentation du prototype du logiciel de gestion est en langage Java.

Cette méthode crée une hiérarchie des tests à effectuer en fonction des projets, du logiciel et de ses multiples variantes de paramètres utilisés. La méthode prévoit la possibilité de lancer en séquence les tests d'un logiciel ou de lancer simultanément des logiciels avec chacun leurs tests. De même, elle permet de simplifier la création de fichiers étalons et elle inclut plusieurs options qui gèrent leur comparaison avec des fichiers textuels ou binaires. Une de ces options est particulièrement intéressante. Il s'agit de la possibilité d'inclure des outils qui peuvent servir à valider les fichiers de sortie ou encore transformer un fichier binaire en fichier texte pour visualisation. La méthode prévoit la transmission de rapports de tests aux divers intervenants dans l'organisation de l'entreprise de même que l'archivage de ces rapports.

La méthode proposée se veut une nouvelle approche simple et réaliste au problème de la gestion des tests dans les très petites équipes de développement informatique.

REMERCIEMENTS

Je remercie tout d'abord M. Sylvain Boivin, mon directeur, de m'avoir fourni l'opportunité de faire cette maîtrise; ses suggestions et orientations lors de la rédaction de ce document étaient bien plus à propos que je ne saurais l'avouer.

Un merci tout spécial à M. Luc Morin qui m'initia à la méthode des logiciels de tests et à la problématique des très petites équipes de développement.

De plus, l'accomplissement de ce mémoire de maîtrise aurait été beaucoup plus difficile sans l'aide précieuse de Mme Caroline Brassard et de M. Éric Larouche. Leurs discussions, conseils, suggestions ou simplement leur oreille attentive m'ont été précieux.

Un merci spécial à tous les oubliés de l'UQAC : les gardiens (pour mes clefs de bureau oubliées), les gens du ménage (pour mes craies de tableaux et leur sourire aux heures tardives) en passant par les secrétaires.

Pour terminer, je remercie MM. Jean Rouette et Richard Bouchard pour leur patience et leur compréhension (sans qui les méandres bureaucratiques m'auraient paru plus difficiles que cette maîtrise).

En bref, comme le dit le proverbe africain : « Il faut un village pour élever un enfant ». Et ce village était l'UQAC.

TABLE DES MATIÈRES

RÉSUMÉ ii
REMERCIEMENTS iii
TABLE DES MATIÈRES iv
LISTE DES FIGURES vi
LISTE DES TABLEAUX vii
LISTE DES EXEMPLES viii
LISTE DES ANNEXES ix
INTRODUCTION 1
1 THÉORIES ET CONCEPTS 5
1.1 Méthodes de développement de logiciel 5
1.1.1 Méthodes planifiées 5
1.1.2 Méthodes légères 9
1.1.3 Remarques 20
1.2 Processus de test 21
1.2.1 Définitions 23
1.2.2 Domaines de tests 25
1.2.3 Test a priori 28
1.2.4 Test a posteriori 32
1.2.5 Test de régression 34
1.3 Méthodes de tests 38
1.3.1 Boîte blanche 39
1.3.2 Boîte noire 39
1.3.3 Boîte grise 39
2 MÉTHODES USUELLES DE TESTS 40
2.1 Conditions aux frontières 41
2.2 Force brute 47
2.3 Injection volontaire de fautes logicielles 48
2.3.1 Injection dynamique 48
2.3.2 Injection statique (mutation de code) 48
2.4 Génération aléatoire des tests 50
2.5 Génération aléatoire des données 52
2.6 Génération automatique des données 53
2.7 Patrons de tests 56
2.8 Logiciel de tests 57
2.9 Gestion des erreurs 58
2.10 Revue de code 59

2.11 Tests d'interfaces graphiques 60
2.12 Logiciels types 61

3 MÉTHODE GÉNÉRIQUE DE GESTION DES TESTS D'UNITÉS LOGICIEL 65

3.1 Description de la méthode de gestion 66
3.1.1 Environnement 66
3.1.2 Éléments de la méthode 68
3.1.3 Résultats des tests 70
3.1.4 Spécifications du logiciel de tests 71
3.1.5 Structure de répertoires 76
3.2 Preuve de concept 78
3.2.1 Implémentation : mode console 78
3.2.2 Implémentation : mode graphique 83
3.2.3 Base de données 85
3.3 Prototype de la méthode proposée 87
3.3.1 Séquence d'opérations 87
3.3.2 Comparaison fichiers de sortie/étalon 96
3.3.3 Rapport de tests 98
3.3.4 Code source, binaire et documentation 99
3.4 Expérimentation in situ 101
3.4.1 Modèle du projet 102
3.4.2 Tests de modules 104
3.4.3 Contraintes et solutions 108
3.5 Discussion 111
3.6 Développements futurs 113

CONCLUSION 115

BIBLIOGRAPHIE 116

ANNEXE 128

LISTE DES FIGURES

Figure Page

1 : Développement en cascade ... 6
2 : Développement en V ... 7
3 : Développement en spirale ... 8
4 : Développement par prototype ... 12
5 : Développement *Open Source* (phase implémentation) ... 18
6 : Développement *Open Source* (phase fabrication/distribution) ... 19
7 : Développement *Open Source* (phase utilisation/tests) ... 20
8 : Processus de test de logiciel ... 22
9 : Hiérarchie des domaines de tests ... 26
10 : Conception d'un test a priori ... 29
11 : Construction d'un test a priori ... 30
12 : Méthode pour générer un fichier étalon ... 31
13 : Test a posteriori (boucle rétroaction) ... 33
14 : Structure des unités logiciel ... 35
15 : Conditions aux frontières ... 42
16 : Conditions aux frontières (D = [6, 10]) ... 46
17 : Cohésion entre les environnements ... 67
18 : Architecture d'un logiciel de tests ... 71
19 : Écriture de fichiers de sortie (Script) ... 79
20 : Comparaison fichiers de sortie versus fichiers étalons ... 81
21 : Comparaison fichier rapport versus fichier rapport-étalon ... 82
22 : Interface du prototype de la preuve de concept ... 84
23 : Schéma entité relation des données ... 86
24 : Dépendance du logiciel de gestion ... 88
25 : Création d'un projet de tests ... 90
26 : Valider les résultats du test ... 91
27 : Valider un test de régression ... 93
28 : Comparer autrement ... 94
29 : Relations du logiciel de gestion avec les logiciels outils ... 95
30 : Modèle du projet ... 102
31 : Test du protocole de communication ... 105

LISTE DES TABLEAUX

Tableau Page

1 : Structure incrémentale des tests de régression 36
2 : Exemples de domaine de valeurs d'entrée 41
3 : Conditions aux frontières 43
4 : Dépassement de capacité 44
5 : Spécifications de la fonction de validation du mot de passe 45
6 : Chaîne test de conditions aux frontières 47
7 : Résultats des tests (température de l'eau) 49
8 : Résultats tests (température de l'eau, après mutation de code) 50
9 : Matrice de transition 51
10 : Comparaison couverture de code 54
11 : Logiciels de tests 62
12 : Description de champ 76
13 : Structure de répertoires suggérée 77
14 : Comparaison de jetons 97
15 : Solutions proposées face aux contraintes de tests 109

LISTE DES EXEMPLES

Exemple Page

1 : Conditions aux frontières 43
2 : Additionneur 47
3 : Eau ne bout pas 49
4 : Génération aléatoire des données (cas problème) 52
5 : Chemin d'appel des fonctions 59
6 : Implémentation avec jUnit 63
7 : Fichier d'entrée (type simple) 74
8 : Fichier de tests (type simple : bis) 74
9 : Fichier de tests (type complexe) 75
10 : Fichier script 80
11 : Script (comparaison de fichier) 81
12 : Rapport de tests (partiel) 82
13 : Fichier verdict.txt 83
14 : Rapport de tests conformes 98
15 : Rapport de tests non-conforme 99
16 : Activation mode console 100

LISTE DES ANNEXES

Annexe | Page

1 : Glossaire 120
2 : Logiciel de tests 122
3 : Fichier script 125
4 : Fichier de données XML 128

INTRODUCTION

Selon Chuck Reid [1] (traduction) : *En théorie, il n'y a pas de différence entre la théorie et la pratique; en pratique, il y en a.*

C'est en cherchant à prouver que cette assertion est fausse que les programmeurs sont devenus des testeurs. Ceux-ci cherchent donc à montrer que le logiciel fait en pratique ce qu'il doit faire en théorie.

Dans le développement de logiciel, les processus et les techniques qui montrent ce que le logiciel fait ce qu'il est supposé faire et seulement ce qu'il est supposé faire se nomme « phase de tests de logiciel ». La phase de tests de logiciel fait partie intégrante de tout processus de développement de logiciel, car seuls les tests permettent de valider la correspondance entre ce que doit faire le logiciel et ce qu'il fait réellement.

Il est bien documenté [2;3] que les coûts des tests logiciel comptent pour environ 50 % du coût total du développement et même plus dans les systèmes critiques. Les très petites entreprises de programmation ne peuvent, au point de vue financier, déléguer un programmeur spécifiquement pour la conception et l'exécution des tests. Dans ces entreprises, et même dans les entreprises plus grandes, la phase de tests est souvent reléguée vers la fin du projet avec tous les aléas que cela comporte : tests bâclés ou non pertinents, découverte de fautes majeures et autres semblables.

Il existe de nombreux outils informatiques qui permettent de gérer les tests, à partir du gratuiciel jusqu'au système corporatif valant jusqu'à 150 000 $. À un tel prix, il est certain que les très petites entreprises ne peuvent acheter le système corporatif. Quant aux gratuiciels, ils sont généralement limités ou ont leurs propres particularités qui peuvent causer des difficultés aux petites entreprises. Par exemple, outil à plate-forme unique, outil à langage unique ou impossibilité d'ajouter des tests sans recompiler le code source.

Pour satisfaire les besoins de qualité de logiciel et de flexibilité des très petites entreprises, un système de gestion des tests devrait :

1. Déléguer aux programmeurs les tâches reliées à la phase de tests;
2. Pouvoir transférer les connaissances de tests d'un langage de programmation à un autre;
3. Avoir un outil multi plate-forme de gestion;
4. Se transporter aisément d'un ordinateur à l'autre;
5. Permettre d'étendre la couverture des tests sans recompiler le binaire;
6. Requérir peu de formation;
7. Utiliser les tests de régression[1];
8. Grouper les tests par logiciel, par groupe de logiciels ou par projet;
9. Permettre l'exécution automatique de tests à des moments ou des fréquences prédéterminés;
10. Générer des rapports d'exécution de tests;
11. Transmettre les rapports de tests à la personne responsable du composant testé.

Sous ces contraintes, les tests de régression sont l'élément centralisateur et simplificateur qui permet à un groupe restreint de programmeurs de fonctionner en s'assurant que le travail effectué précédemment est conforme en regard des spécifications du logiciel. Pour automatiser la gestion des tests de régression, il est nécessaire que le logiciel sous test soit déterministe c'est-à-dire qu'il présente les caractéristiques suivantes [4] :

1. Il génère correctement[2] la sortie du logiciel;
2. Avec les mêmes données à l'entrée, le logiciel génère la sortie correcte à chaque exécution;
3. Toutes les variantes de sortie du logiciel pour des données spécifiques à l'entrée doivent être correctes;
4. Les variantes dans les valeurs de sortie sont bornées.

---

[1] Test de régression : répétition de tests pour s'assurer que les modifications apportées n'ont pas introduit de faute dans un logiciel préalablement correct (voir section 1.2.5).

[2] Ici, le mot « correctement » désigne le fait que la sortie du logiciel est conforme aux critères de la spécification du logiciel.

Par exemple, un logiciel qui fait la somme de deux nombres particuliers en entrée retournera toujours la somme de ces deux nombres à chaque exécution. Il est donc déterministe.

Un logiciel qui ne satisfait pas cette définition est dit non déterministe. Usuellement, il s'agit d'un logiciel qui interagit avec un autre logiciel. Le moyen de communication entre ces logiciels peut être une pile d'événements ou un réseau. Les piles d'événements sont des moyens de communication utilisés par les systèmes d'opération (par exemple : Windows, Linux, QNX, etc.) pour gérer l'ordre de réception des messages événementiels. Un réseau est un ensemble d'ordinateurs reliés entre eux dans le but d'échanger des données.

De par la nature même de ces moyens de communication, il est possible, et même probable, que le temps de transfert entre les deux logiciels soit différent d'une fois à l'autre. Une des causes immédiates est l'encombrement de ces moyens de communication.

Un exemple typique de logiciel non déterministe est un logiciel mesurant le temps de transfert d'un fichier sur un réseau informatique, car il est probable que le temps de transfert sera variable d'expérimentation en expérimentation, en fonction de l'activité du réseau. Toutefois, cette valeur peut être bornée et localisée dans un fichier de sortie selon une syntaxe. Il est aussi possible de poser des conditions sur cette valeur, par exemple, le temps de transfert doit être entre 0 sec et 20 sec.

Le but du présent projet est de concevoir et d'implémenter, sous forme de prototype, une méthode générique de gestion des tests de régression applicable dans divers environnements logiciels. Pour atteindre ce but, les livrables suivants seront générés :

1. Description de la méthode de gestion des tests;
2. Logiciel de gestion des tests (prototype);
3. Exemple d'implantation d'un logiciel de tests;
4. Base générique de données pour la gestion des tests.

Ce mémoire de maîtrise est organisé en cinq sections distinctes. L'introduction expose le problème auquel fait face la petite entreprise en informatique en rapport avec la gestion des tests et précise les livrables qui seront effectués. Le chapitre 1 : « théories et concepts» montre quelques-unes des méthodes de développement de logiciels et la place des tests dans celles-ci. Le chapitre 2 : « méthodes usuelles de tests» présentera les processus de tests logiciels ainsi que des méthodes d'inspection et de mesure. Suivra le chapitre 3 : « méthode générique de gestion des tests d'unités logiciel » qui développe une solution générique au problème de gestion de tests d'unités logiciel pour les petites entreprises. Finalement, la conclusion sera présentée.

Dans le présent document, si un élément du code informatique est requis dans le texte, celui-ci est en caractère gras, par exemple, **int a = 0**;. De même, lorsqu'un mot du glossaire est rencontré pour la première fois dans le texte, celui-ci possède un astérisque, par exemple, Fichier étalon*. Les citations ainsi que les termes anglais sont en caractère italique, par exemple, *framework*. Les mots soulignés indiquent une importance portée à ceux-ci, par exemple : plus.

1

THÉORIES ET CONCEPTS

Ce chapitre contient les éléments pour établir les connaissances nécessaires pour bien situer les divers types de processus de développement de logiciels des petites entreprises et la position que les tests occupent au sein de ceux-ci. Tout d'abord, les principales méthodes de développement logiciel sont présentées. Suivent des exposés sur le processus de test, les méthodes de tests et, pour terminer, sur les métriques du code informatique.

1.1 Méthodes de développement de logiciel

Une méthode de développement de logiciel est l'ensemble des étapes de fabrication de logiciel en partant de la définition de la problématique jusqu'à l'acceptation de la solution logicielle par celui-ci.

Dans cette section, quelques méthodes de développement de logiciel bien connues seront présentées, des plus anciennes (cascade et en V) aux plus récentes (*Feature Driven Development* et *XP*) en prenant soin de mettre en évidence la place des tests dans celles-ci. Il est important de bien comprendre les méthodes de développement de logiciel car elles forment le contexte dans lequel le processus de test s'insère.

1.1.1 Méthodes planifiées

Une méthode planifiée — la littérature anglophone utilise la locution : *plan driven development* [5;6;7] — est définie comme une méthode où le processus de développement est strict et suit des étapes bien déterminées et séquentielles. La

première méthode présentée est la plus connue. C'est la méthode dite *Waterfall* ou méthode en cascade. Suivra la méthode de développement en V. Cette section se terminera par la méthode de développement en spirale.

1.1.1.1 Développement en cascade

Cette méthode est basée sur une vision rectiligne du développement de logiciel. La figure 1 montre les étapes menant à la réalisation d'un projet avec ce mode de développement.

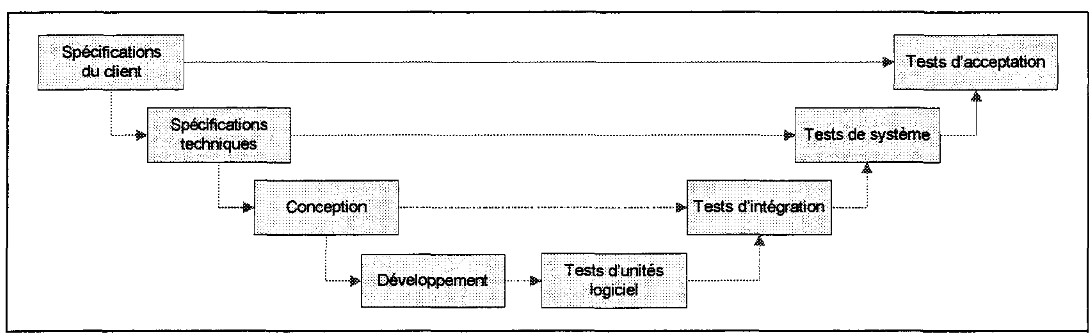
Figure 1 : Développement en cascade

Selon Marick [8], cette approche du développement logiciel n'est plus adaptée aux besoins d'affaires d'aujourd'hui. Le plus grand problème de cette méthode est que chaque étape doit être terminée avant de passer à la suivante. Un autre problème important est que les tests en sont absents.

1.1.1.2 Développement en V

L'étape suivante dans l'évolution des processus de développement est le développement en V. Il s'agit essentiellement d'un développement en cascade auquel sont ajoutés des tests.

Figure 2 : Développement en V



Dans cette méthode, les tests sont séparés en quatre parties :

1. Tests d'unités logiciel : valide que l'implémentation soit conforme à la conception;
2. Tests d'intégration : valide que les unités logiciel s'intègrent les unes aux autres;
3. Tests de système : valide que le système complet rencontre les spécifications techniques;
4. Tests d'acceptation : valide que le système complet rencontre les spécifications du client.

Cette rigidité dans la hiérarchie des tests s'accommode mal de la vitesse des changements dans les règles d'affaires ou dans les spécifications d'un logiciel. Des variations de cette méthode sont donc apparues pour permettre une plus grande adaptabilité au marché d'affaires. Selon cette ligne de pensée, la méthode de développement la plus efficace semble être la méthode dite « cascade itérative ». Cette méthode consiste à prendre de plus petits incréments de spécification et à itérer chaque incrément sur un cycle en V. Ces petits incréments influent sur toute la chaîne de développement, accélérant ainsi le processus de génération de produits* livrables à valeurs ajoutées pour le client.

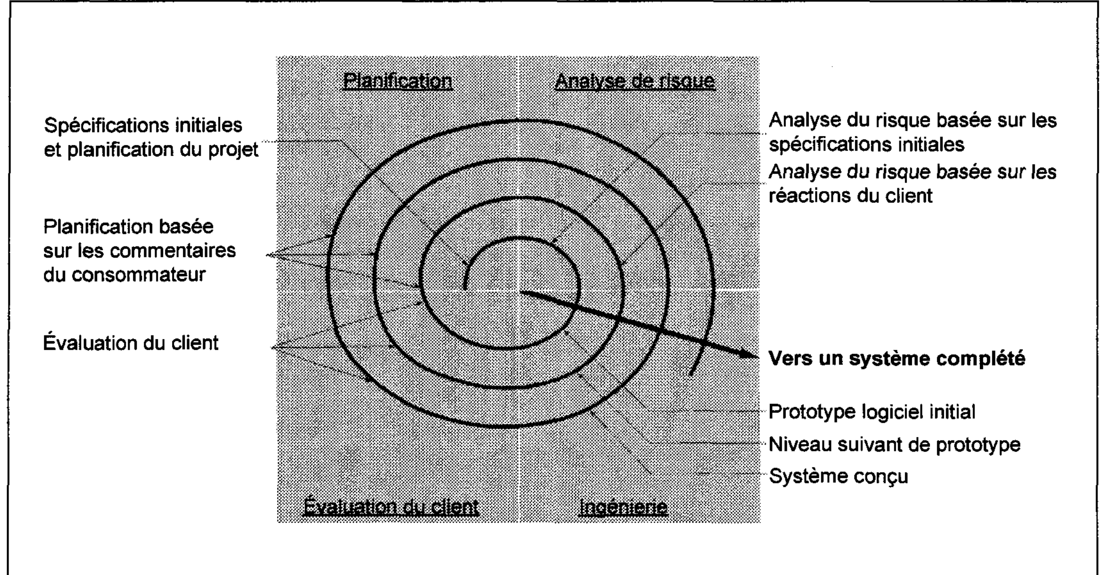
1.1.1.3 Développement en spirale

La méthode de développement en spirale [9] *regroupe les meilleures* caractéristiques des méthodes de développement en cascade et par prototype (section 1.1.2.1 : Développement par prototype) tout en y ajoutant un nouvel

élément : l'analyse du risque. Le modèle, représenté par la figure 3, définit quatre activités majeures :

1. Planification : déterminer les objectifs, les alternatives et les contraintes;
2. Analyse de risques : analyser les alternatives et identifier/résoudre les risques;
3. Ingénierie : développer le produit;
4. Évaluation du client : valider les résultats de la phase ingénierie.

Figure 3 : Développement en spirale



L'implémentation* de cette méthode demande une grande compétence et devrait être limitée aux projets innovants en raison de l'importance qu'elle accorde à l'analyse des risques.

Dans cette méthode, les tests informatiques sont localisés dans deux sections :

- Ingénierie : tests de débogage effectués durant la fabrication du prototype. Ces tests peuvent être rigoureux ou simplement des tests de fonctionnement positif;
- Évaluation du client : valide que le prototype exécute bien ce qui était mentionné dans les spécifications. Dans le cas d'une anomalie, celle-ci est alors inscrite dans les spécifications pour la prochaine itération ou une itération ultérieure.

La méthode de développement en spirale, contrairement à la méthode de développement par prototype, prévoit de garder les prototypes à la fin du cycle de développement. Toutefois, d'autres tests non informatiques sont effectués dans les autres phases de la méthode :

- Spécifications : valider que la documentation décrive clairement ce que le client spécifie;
- Analyse du risque : valider que les informations, les calculs et les conclusions soient conformes. Valider que les méthodes utilisées pour l'évaluation des risques soient conformes aux règles de l'entreprise, aux meilleures pratiques ou aux règles de l'art.

Un aspect important de ce modèle est que plus la spirale fait d'itérations, plus le logiciel est complet. Cependant, à chaque itération, une analyse du risque est requise. Le fait de définir de façon formelle quels sont les risques et comment les résoudre ou les contourner est très lourd. C'est principalement ce facteur qui place la méthode de développement en spirale dans le domaine des méthodes lourdes.

1.1.2 Méthodes légères

Par comparaison aux méthodes lourdes, les méthodes légères sont basées sur le principe de livraisons rapides et régulières du logiciel avec valeur ajoutée. De plus, elles mettent l'accent sur l'utilisation systématique des tests de régression (section 1.2.5 : Test de régression). Cette classe de méthodes de développement, dite Agile, se distingue par des principes simples qui ont été rédigés sous forme d'un manifeste [10] dont voici un résumé :

1. Accepter volontairement les changements aux spécifications, même tard dans le projet;
2. Livrer fréquemment le logiciel fonctionnel;
3. Les gens d'affaires et les développeurs doivent travailler ensemble quotidiennement tout au long du projet;
4. Construire le projet autour de personnels motivés;

5. La façon la plus efficace de communiquer avec l'équipe de développement est en ayant des conversations face-à-face;
6. Un logiciel fonctionnel est la mesure primaire du progrès;
7. Promouvoir un développement constant et durable par l'équipe de développement;
8. Attention continue à l'excellence technique et à la bonne conception;
9. La simplicité – l'art de maximiser la somme de travail non fait – est essentielle;
10. L'équipe analyse ses méthodes de travail et cherche à les améliorer.

Un facteur émane de l'approche Agile : l'importance donnée à l'équipe de développement. L'organisation des équipes peut sembler chaotique pour certains, mais les individus possèdent des méthodes de travail cohérentes. Conséquemment, cette approche devrait bien fonctionner avec des programmeurs expérimentés. Cependant, il est à craindre que dans le cas où l'équipe de développement est composée de programmeurs débutants ou non formés aux méthodes de développement récentes, le chaos s'installe.

De plus, certains propagateurs de l'approche Agile sont convaincus que les anciennes méthodologies sont obsolètes. Cependant, Berard [11] tente de montrer que de nombreux points de divergence, cités par des « *Agile Zealots* », entre les approches de développement Agile et les méthodes classiques sont en fait des miroirs aux alouettes et de fausses conceptions qui ne résistent pas à l'analyse.

Dans toutes les méthodes de cette approche du développement, les tests sont le fondement de l'item n°6 présenté dans la liste précédente car seuls les tests peuvent prouver que le logiciel est fonctionnel. Cependant, cette approche ne précise pas explicitement la localisation des tests dans le schéma de développement. Toutefois, il est possible de déduire que, comme dans toutes les méthodes de développement, les tests sont localisés lors du développement, de l'assemblage et de l'acceptation par le client. Une attention particulière doit être placée sur la gestion des tests.

Il existe plusieurs méthodes dites Agile [12], toutefois seules quelques-unes seront présentées. De plus, le développement par prototype a été inclus dans la famille des méthodes Agile car cette méthode de développement est axée sur le développement d'un produit avec peu d'importance donnée à la documentation des spécifications du logiciel.

1.1.2.1 Développement par prototype

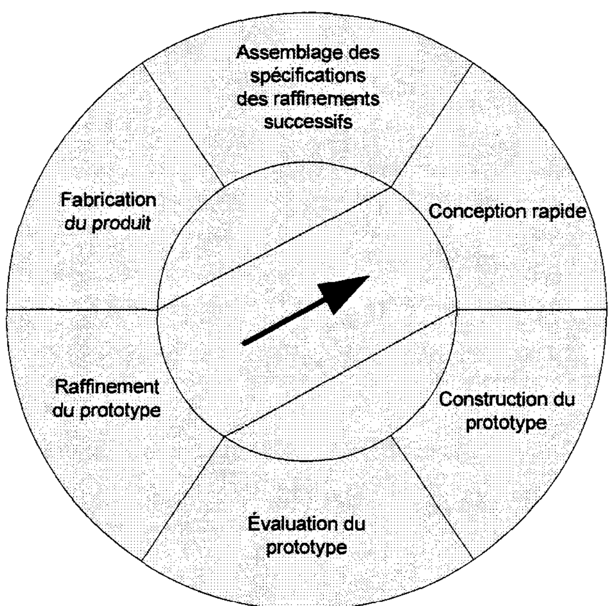
Souvent, le client ne possède que des objectifs généraux pour son logiciel et les spécifications du logiciel (c'est-à-dire : les entrées/sorties, les interfaces Humain-Machine, les règles d'affaires, etc.) ne sont pas encore définies. Dans d'autres cas, le programmeur peut avoir des doutes sur la performance d'un système d'opération imposé ou sur la qualité d'un algorithme ou d'une règle d'affaires. Ces quelques facteurs sont suffisants pour tenter d'obtenir une plus grande précision dans les spécifications du logiciel. Le développement par prototype peut alors être une façon rapide et simple d'atteindre ces buts par la construction d'un modèle. Ces prototypes peuvent avoir les formes suivantes [13] :

- Un modèle papier qui permet de montrer le fonctionnement de l'interaction entre l'humain et la machine;
- Un logiciel utilisant les fonctionnalités, les algorithmes et l'interface avec des données fictives qui proviennent du code. Par exemple : un logiciel d'interrogation de base de données simulera celle-ci par l'utilisation de quelques données directement dans le code du logiciel. L'effet visuel serait « comme si » le logiciel était branché à la base de données;
- Un prototype opérationnel qui fonctionne avec seulement un sous-ensemble des fonctionnalités;
- Un logiciel déjà existant et qui implémente une partie des fonctionnalités désirées.

Dans les cas où il y a construction d'un prototype logiciel, la méthode de développement par prototype prévoit que le logiciel sera mis de côté à la fin du cycle de développement. Il est important de se rappeler que le but du développement par prototype est de préciser les spécifications du logiciel et non

d'avoir un logiciel répondant aux spécifications. La figure 4 montre les étapes du développement d'un prototype.

Figure 4 : Développement par prototype



Le processus de développement par prototype débute par les spécifications du logiciel. Une conception rapide est alors exécutée, puis le prototype est construit. À ce moment, le client évalue le prototype puis précise les spécifications du logiciel. Un processus d'itération s'en suit et le prototype est raffiné, passant de nouveau dans les phases de conception rapide et de construction du prototype pour de nouveau être évalué par le client. Lorsque le client est satisfait et que les spécifications sont suffisamment précises, la vie utile du prototype est terminée. Les spécifications sont alors utilisées pour construire le logiciel selon la méthode de développement désirée.

Idéalement, ce paradigme de développement sert à explorer et à définir les spécifications du logiciel. Si le prototype est réellement fonctionnel, il arrive parfois que le client décide de garder celui-ci. Cette décision de garder ou non le prototype doit être mûrement pesée car celui-ci possède les faiblesses du

prototype : conception insuffisante, insuffisamment testé, code spaghetti*, très faible gestion des erreurs, peu efficace, etc.

Dans cette méthode de développement, les tests du logiciel sont essentiellement les tests de débogage du programmeur (voir section 1.2.1.2 Débogage). C'est-à-dire seulement les tests qui peuvent confirmer le fonctionnement positif du produit, puisque la validation des données est minimisée. Il est à remarquer que dans cette méthode, les tests de logiciel sont minimisés car ils ne servent qu'à montrer au développeur que son code informatique fonctionne correctement.

La méthode de développement par prototypage fait partie des méthodes légères car elle est très flexible et minimise les documents écrits au long du projet. En effet, c'est à partir du prototype que les spécifications du logiciel sont écrites.

1.1.2.2 Test Driven Development

Selon Kent Beck [14;15], Janzen et Saiedian [16] et la compagnie AdaptionSoft [17], le *Test Driven Development (TDD)* est un processus de développement Agile basé sur la conception et l'écriture des tests avant l'implémentation du code. Cette étape est suivie de l'automatisation de l'exécution de ces tests. Idéalement, avec *TDD*, il n'y a pas de débogage mais du pré-débogage. Les bogues sont trouvés et éliminés à l'implémentation . Les tests doivent gérer les deux cas principaux, soit :

- Test de fonctionnement positif : la fonction **fait** ce qu'elle est censée faire;
- Test de fonctionnement négatif : la fonction **ne fait pas** ce qu'elle est censée faire. (Si un test de fonctionnement n'est pas positif, il est nécessairement négatif.).

Un test de fonctionnement positif est un comportement attendu en tenant compte des intrants dans la fonction. Voici deux exemples permettant de l'illustrer :

1. Une fonction `int somme(x,y)` reçoit deux valeurs et retourne la somme de ces deux valeurs : `x + y` ;
2. Une fonction **`division(x,y)`** reçoit deux valeurs et retourne la division de ces deux valeurs : **`x/y`**. Dans le cas d'une division par zéro, le comportement est fonction du langage utilisé et du comportement désiré par les spécifications[3].

TDD est un processus de développement qui s'imbrique bien avec le principe de tests de régression (section 1.2.5 : Test de régression). Le bénéfice le plus clair est la possibilité de valider le système à chaque étape de son élaboration, car il y a toujours un ensemble de tests qui protège constamment la validité du système. Ces tests permettent de trouver et d'éliminer les bogues aussitôt qu'ils sont trouvés.

1.1.2.3 Feature Driven Development

Le *Feature Driven Development* (*FDD*) [18] est un autre processus Agile qui peut être assimilé de premier abord comme une version hybride du processus en cascade et du processus *TDD*. Tout d'abord, le processus *FDD* recense toutes les fonctionnalités que le système doit posséder en une liste. Celle-ci est ordonnée selon des critères déterminés par le client et l'équipe de développement. Les descriptions des fonctionnalités sont complètes et suffisantes pour développer un modèle global, sans toutefois entrer dans les détails. La liste de fonctionnalités possède une double fonction : aider à construire un plan de travail et servir de feuille de route pour l'avancement des travaux, conformément au principe de Machiavel [19] : « Diviser pour régner ». C'est au moment de la réalisation d'une

[3] Notons qu'en programmation objet, en cas d'erreur la fonction pourrait lancer « une exception » qui serait alors captée par la fonctionnalité « `try{...} catch{...}` » du langage. Dans un langage procédural, tel le **C**, advenant un fonctionnement incorrect d'une fonction, une variable globale pourrait être mise en « code d'erreur ». Un bel exemple de ceci est la variable « `errno` » en langage **C**.

fonctionnalité que le modèle de tests *TDD* (voir section 1.1.2.2 : Test Driven Development) peut être appliqué.

1.1.2.4 eXtreme Programming

En 1996, Kent Beck [15] a introduit la méthode *eXtreme Programming* (*XP*). Celle-ci est devenue la méthode la plus connue parmi les méthodes Agile. Son but est simple et bien exprimé par la locution anglaise : *eXtreme Programming = Extreme Business Value.* Donc, *eXtreme* est pris dans le sens de toujours maximiser la désirabilité du code informatique implémenté. Ce type de construction « à la pièce » est basé sur 12 principes simples [20;21;22;23] :

1. *Planning Game* : planifier la prochaine version distribuée en combinant les priorités d'affaires et les estimations techniques;
2. *Small Releases* : les incréments des versions « développement » du logiciel sont minimes. Habituellement, une version « production » contiendra plusieurs versions « développement »[4];
3. *Metaphor* : l'utilisation de métaphores montrant comment le système complet devrait fonctionner guide le développement;
4. *Simple Design* : une analyse, simple, sinon sommaire, est effectuée et les programmeurs commencent à travailler avec celle-ci;
5. *Testing* : les programmeurs écrivent continuellement des tests qui doivent fonctionner entièrement et parfaitement avant de continuer le développement. Le client écrit des tests qui lui permettront de démontrer que la fonctionnalité est terminée;
6. *Pairing* : travailler par paire de programmeurs sur une seule machine;
7. *Refactoring* : méthode de travail qui consiste à améliorer la conception lorsque le programmeur revient sur du code déjà programmé;
8. *Integration* : l'intégration des divers composants* ou bibliothèques doit être continue et validée à chaque livraison de version de production ou de développement;
9. *Shared Code* : l'appartenance du code est collective, chaque paire de programmeurs peut améliorer tout code n'importe quand;
10. *Work Pace* : la cadence de travail est telle (par exemple : 40 heures/semaine) qu'elle pourrait être soutenue indéfiniment. Autrement dit,

[4]Note : une version de développement pourrait s'échelonner sur une période allant de 3 à 6 semaines, alors qu'une version de production, une nouvelle version distribuable comprenant plusieurs versions de développement, pourrait être disponible tous les 6 mois.

des heures régulières et constantes; les heures supplémentaires sont ponctuelles et sont considérées comme une exception et non comme une règle;

11. *On-site custumer* : inclure un usager dans l'équipe de développement disponible en tout temps pour répondre aux questions de l'équipe de développement;
12. *Code Standard* : l'utilisation de règles de programmation, au niveau du format du code, permet à chaque programmeur de se retrouver plus facilement dans du code qui n'est pas le sien.

Une des principales oppositions à l'utilisation de *XP* est le coût engendré par la programmation en paires. Cependant, selon les expériences de Cockburn et William [24], la programmation en paires est économiquement viable et génère des produits de meilleure qualité. Toutefois, cette expérimentation de programmation par paires était académique et la preuve de son applicabilité dans le monde du travail reste en suspens.

Les tests dans cette méthode de développement sont basés sur la méthode de développement *Test Driven Development* (section 1.1.2.2 : Test Driven Development). Van Deursen, Moosen, van den Bergh et Kok [25] ont exposé des comportements particuliers en ce qui a trait au *refactoring* du code de tests. Ceux-ci apparaissent lors de l'utilisation de xUnit (section 3.11 : Logiciels types) et lorsqu'il y a un grand nombre de tests.

1.1.2.5 Développement « Open Source »

Le développement *Open Source* se démarque des autres méthodes de développement. En effet, c'est une approche radicalement différente qu'il convient de traiter à part. Raymond [26] , un des initiateurs du mouvement *Open Source*, en explique le mécanisme de fonctionnement par l'allégorie de la cathédrale et du bazar. Celle-ci consiste à assimiler le développement de logiciel par les méthodes dites conventionnelles (cascade, *XP*, *FDD*, etc.) à la construction d'une cathédrale : propre, prévisible, ordonnée. À l'opposé, la construction d'un bazar est assimilée un assemblage de morceaux divers qui tendent vers un même but

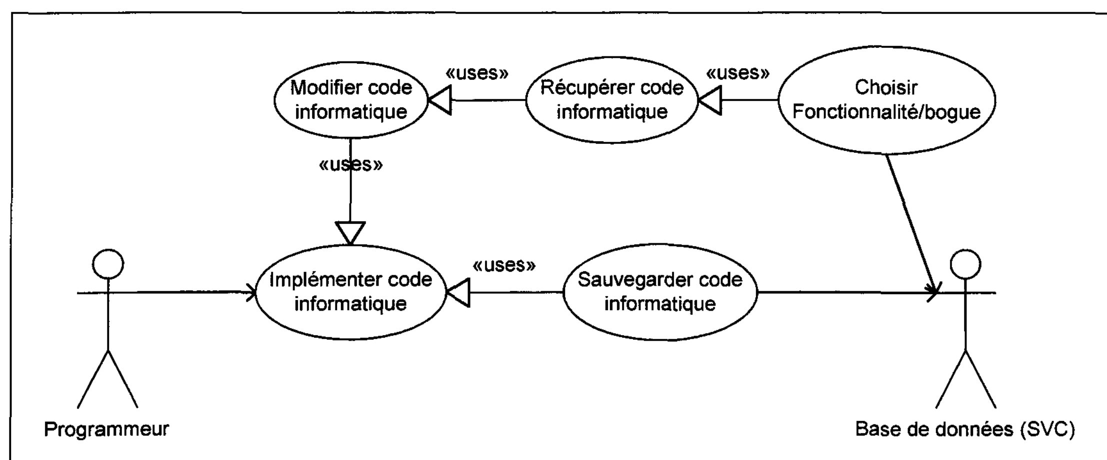
(développement *Open Source*). Ce bazar semble hétéroclite à première vue. Cependant, il fonctionne ! Selon Raymond [27], pour que cette méthode de développement puisse être appliquée avec succès, un logiciel doit avoir :

1. Un nombre suffisant de fonctionnalités pour être utile à l'utilisateur potentiel;
2. Une qualité permettant son utilisation;
3. Une liste des fonctionnalités futures à implémenter*;
4. Des outils de communication et des standards de programmation[5].

En outre, le lancement public d'un prototype initial suffisamment complet pour être utile à une large base d'utilisateurs est un facteur déterminant. Le développement *Open Source* est exécuté sur une base volontaire, chaque programmeur étant libre de développer la fonctionnalité qu'il désire. Minimalement, lors de l'ajout d'une fonctionnalité, le programmeur s'assure que le logiciel est opérationnel. Les bêta-testeurs utiliseront le logiciel et, par le fait même, le testeront. C'est l'utilisation intensive de la méthode de tests a posteriori (section : 1.2.4) qui caractérise le plus le développement *Open Source* et fait sa force. Les figures 5, 6 et 7 illustrent l'organisation des phases de développement du modèle *Open Source*.

[5] Le site du groupe SourceForge se spécialise dans la gestion du développement de logiciels *Open Source*.

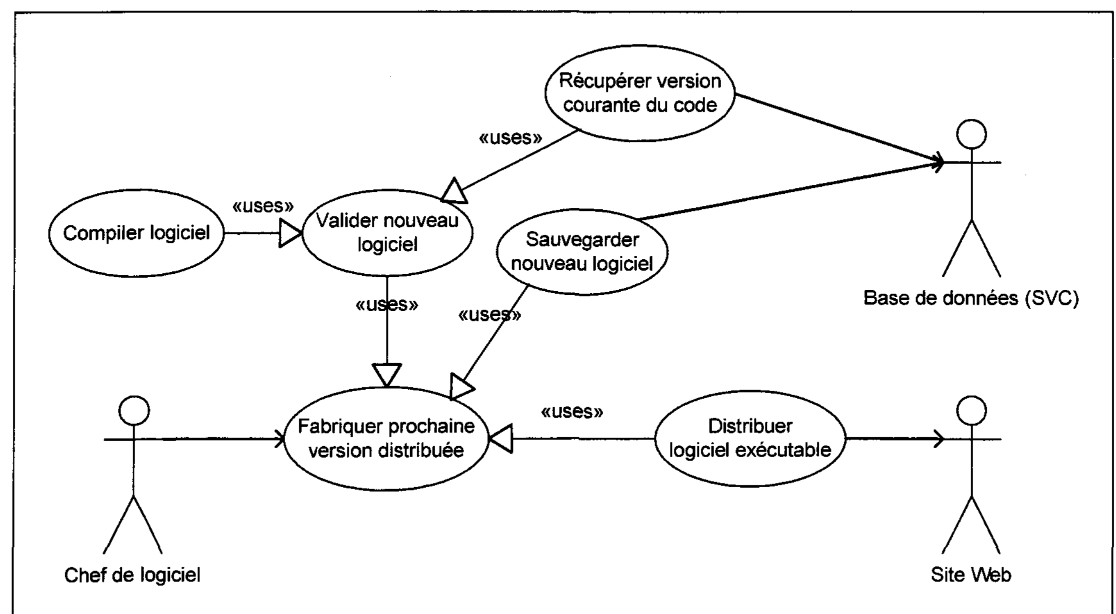
Figure 5 : Développement *Open Source* (phase implémentation)



- Programmeur :
    a) Choisit une fonctionnalité à implémenter ou un bogue à réparer;
    b) Récupère le code informatique;
    c) Effectue le travail (modifier ou implémenter le code);
    d) Sauvegarde le code informatique.
- Base de données (SVC*):
    a) Sauvegarde le code informatique;
    b) Offre les fonctionnalités et les bogues à réparer.

Suite au nouveau code informatique, un nouveau logiciel doit être fabriqué et distribué. La figure 6 montre les différentes actions menant à une nouvelle version du logiciel.

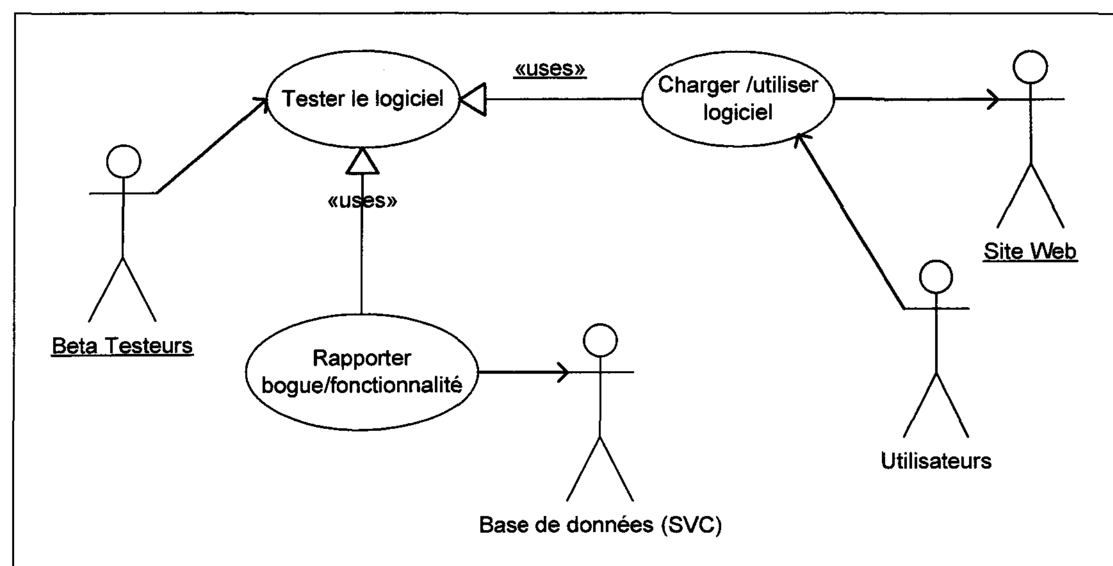
Figure 6 : Développement *Open Source* (phase fabrication/distribution)



- Chef de logiciel :
    a) Récupère le code informatique de la prochaine version;
    b) Valide le nouveau logiciel;
    c) Compile le nouveau logiciel;
    d) Sauvegarde le nouveau logiciel;
    e) Distribue le nouveau logiciel sur le site Web.
- Base de données (SVC) :
    a) Fournit les versions courantes du code informatique pour compiler le logiciel;
    b) Entrepose la version officielle du binaire compilé.
- Site Web :
    a) Reçoit la version distribuable au public et aux bêta-testeurs.

Lorsque la nouvelle version du logiciel est disponible, c'est au tour des bêta-testeurs de l'utiliser et de tester celle-ci, tel que montré à la figure 7. Sur cette figure, les tests sont placés à la fin du cycle de développement. C'est dans cette dernière phase que les bogues ou de nouvelles fonctionnalités désirables sont rapportés dans la base de données SVC. Ces bogues et fonctionnalités sont mis à la disposition des programmeurs dans la base de données SVC et, lorsque sélectionné, un nouveau cycle de test a posteriori débute.

Figure 7 : Développement *Open Source* (phase utilisation/tests)



- Bêta-testeurs :
  a) Charger/utiliser le logiciel;
  b) Tester le logiciel;
  c) Rapporter les bogues ou des fonctionnalités désirées.
- Utilisateurs :
  a) Charger/utiliser le logiciel.
- Base de données (SVC) :
  a) Entreposer les bogues/fonctionnalités pour les fournir aux programmeurs.
- Site Web :
  a) Distribuer la version courante du logiciel.

1.1.3 Remarques

Khramtchenko [28] compare la méthode de développements *Feature Driven Development* avec *XP* et conclut que si les exigences du projet sont plutôt floues au départ, la méthode *XP* est plus adéquate. Dans le cas où les fonctionnalités sont assez bien définies ou si le projet est de grande ampleur, la méthode *Feature Driven Development* est plus adéquate.

Marick [8] a élaboré une méthode de développement basée sur les tests. Sa vision est basée sur deux idées fondamentales. Premièrement, il oriente la

création/exécution des tests par un critère de sélection : « À qui ce produit ferait-il le plus de dommages s'il contenait des bogues ? ». Deuxièmement, le nombre de tests sur les produits livrables devrait être une mesure du développement du logiciel. Plus il y a de tests construits, plus le logiciel est avancé. Ceci semble aller à l'encontre du but d'une entreprise qui est de faire un produit que le client désire. En effet, l'entreprise n'est pas là pour faire des tests, elle est là pour satisfaire son client, quoique l'entreprise utilise les tests pour faire un logiciel qui satisfera son client. Et c'est ce dernier point qui est la force des méthodes Agile : faire des tests pour prouver des fonctionnalités. Autrement dit, les méthodes Agile comptent les fonctionnalités implémentées, et Marick compte les tests générés.

Ainsi, il n'est pas surprenant que l'approche de Marick ait été presque totalement éclipsée par les méthodes Agile. Toutefois, dans un contexte de ressources restreintes, l'idée de Marick de sélectionner les tests en fonction de l'utilisateur qui subirait le plus de dommages d'un bogue est intéressante.

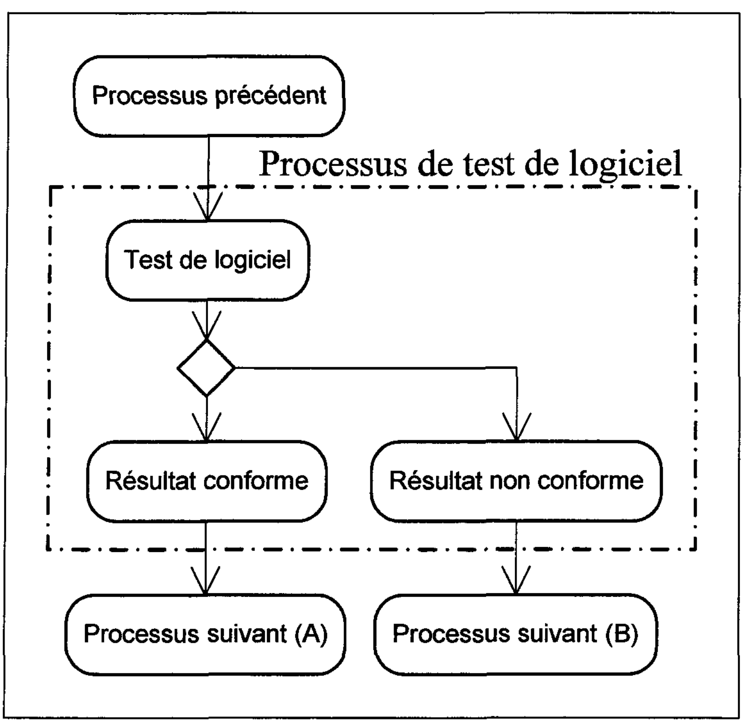
1.2 Processus de test

Selon Hoffman [29] (traduction) :

> *Tester un logiciel est le processus consistant à fournir des entrées au logiciel sous tests et en évaluer les résultats.*

Un test de logiciel est un processus qui fournit des intrants à un logiciel et évalue les résultats. Il est construit à partir des spécifications du logiciel et des critères de sélection des tests de logiciels.

Figure 8 : Processus de test de logiciel



La figure 8 schématise un processus de développement quelconque dans lequel se retrouve un processus de test générique. De façon générale, ce processus de test possède un processus entrant (provenant de « Processus précédent ») et deux processus sortant (« Processus suivant » A et B).

Le test de logiciel est la comparaison du comportement du logiciel avec le comportement attendu de celui-ci. Son résultat peut être conforme ou non. La littérature anglophone [29;30] utilise le terme *Oracle* pour exprimer le comportement attendu.

Hoffman [30] a classé les *Oracles* en cinq catégories ayant chacune leurs avantages et inconvénients :

1. *True Oracle* : génère de façon indépendante des résultats attendus;
2. *Heuristic Oracle* : vérifie quelques valeurs de même que la consistance des valeurs restantes;
3. *Sampled Oracle* : sélectionne une collection déterminée de données d'entrée ou de résultats;

4. *Consistant Oracle* : vérifie les résultats courants avec les résultats antérieurs (test de régression);
5. *No Oracle* : ne valide pas l'exactitude des résultats (seulement des résultats sont produits).

En français, le terme « oracle » signifie une réponse donnée par un dieu à une question qui lui est posée. Cette définition est inadéquate pour ce document. D'autre part, le terme étalon, qui signifie grandeur type servant à définir une unité, est plus adéquat. Il sera utilisé dans le sens où un étalon est une réponse qui a déjà été validée et est correcte. L'étalon servira alors d'objet de comparaison avec les réponses obtenues. Ce document est dédié à la gestion des tests selon un « oracle consistant », item n°4 dans la liste précédente. Dans ce cas, la locution « fichier étalon* » sera utilisée en lieu et place du terme anglais « *oracle* » pour désigner le fichier de comparaison, ceci le distinguant d'un étalon qui, lui, pourrait être un résultat attendu, écrit sur une feuille de papier.

Un aspect important des tests est la possibilité d'optimiser la fabrication et l'exécution de ceux-ci selon les besoins de l'organisation ou du projet. Par exemple, il est possible d'utiliser des critères de sélection pour permettre de prioriser les tests à effectuer et même de déterminer si le test doit être effectué avant la livraison (test a priori) ou après la livraison (test a posteriori). Ces critères de sélection sont variables selon les organisations et les projets et sont hors de la portée de ce document.

1.2.1 Définitions

Il est important de bien séparer la limite entre <u>tests de logiciel</u> (*software testing*), <u>débogage</u> (*debugging**) et <u>assurance-qualité logiciel</u> (*software quality assurance*), ces termes étant parfois confondus dans leur utilisation. Il existe de nombreuses définitions de ces termes [31;32;33;34;35].

1.2.1.1 Tests de logiciel

Certaines des définitions de tests de logiciel sont extrêmement générales et permettent même d'englober les tests des documents papiers ou autres. Voici une traduction de la définition de tests de logiciel selon Berard [32] :

> *Le processus d'examiner quelque chose avec l'intention d'y trouver des erreurs. Quoique les tests de logiciel puissent révéler le symptôme d'une erreur, ils n'en révéleront peut-être pas la cause.*

Cette définition est similaire à la traduction de la définition de l'IEEE [34] pour le terme « testing »:

> *Le processus d'analyser un item du logiciel pour y détecter les différences entre les conditions requises et existantes (les bogues) et d'évaluer les fonctionnalités des logiciels.*

D'autres définitions sont plus contraignantes et se restreignent aux seuls tests qui sont directement reliés au logiciel. Voici la traduction de la définition écrite par Hetzel [35] :

> *Toute activité visant à évaluer un attribut ou une fonctionnalité d'un logiciel ou d'un système et déterminer s'il rencontre les résultats requis.*

Dans le présent document, c'est cette dernière définition de tests de logiciel qui sera utilisée.

1.2.1.2 Débogage

Les définitions de débogage sont peut-être aussi nombreuses que celles de « test de logiciel ». Toutefois, la traduction de la définition de Berard [32] sera utilisée dans le présent document car elle est, en essence, la plus simple :

> *Processus consistant à localiser la cause exacte d'une erreur et à éliminer cette cause.*

Le débogage ne trouve pas l'effet de l'erreur, il trouve plutôt la cause de l'erreur et la corrige. Travaillant avec le processus de test de logiciel, le débogage est un outil puissant pour obtenir un logiciel correct.

1.2.1.3 Assurance-qualité logiciel

Le terme assurance-qualité logiciel est souvent confondu avec test logiciel. Cette confusion provient du fait que certains croient que l'assurance-qualité logiciel sert à mesurer la qualité du logiciel. Cependant, la traduction de l'Institute of Electrical and Electronics Engineers (IEEE) définit l'assurance-qualité [34]:

> *Un ensemble d'activités conçues pour évaluer les processus par lesquels les produits sont développés ou fabriqués.*

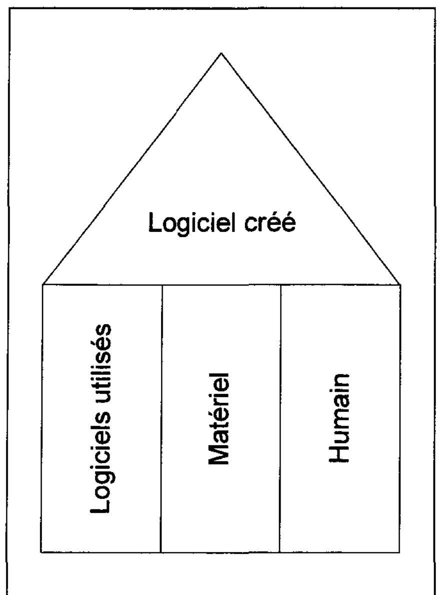
Ce document ne traite pas de l'assurance-qualité logiciel, mais seulement d'un aspect de tout programme qualité, soit les tests de logiciels.

1.2.2 Domaines de tests

Les domaines de tests sont définis comme étant le regroupement d'éléments qui servent à la spécification, l'analyse, la conception, l'implémentation, la validation et la fabrication du logiciel. L'ensemble de ces domaines forme l'univers dans lequel les tests seront conçus et réalisés. Lors du développement d'un logiciel, il existe trois domaines qui peuvent être la cause de non-conformité dans le logiciel créé.

La figure 9 montre les préalables nécessaires pour la création d'un logiciel, auxquels correspondent les domaines de tests.

Figure 9 : Hiérarchie des domaines de tests



Pour qu'un logiciel créé soit conforme aux spécifications du logiciel, il est nécessaire que les domaines « Logiciels utilisés », « Matériel » et « Humain » soient conformes à ces spécifications.

1.2.2.1 Logiciel créé

Le logiciel créé fait l'objet des tests. Chaque test permet de valider qu'une partie des logiciels utilisés, du matériel utilisé sont conformes. Aussi, chaque test permet de valider que la compréhension et l'implémentation des spécifications soient conformes aux spécifications. L'ensemble des tests couvrira donc l'ensemble des parties des spécifications du logiciel créé.

Dans les cas où le logiciel créé doit interagir avec du matériel, celui-ci peut être remplacé par des fichiers de simulation ou par des objets de simulation, c'est-à-dire des objets qui simuleront son comportement (section 2.7 : Patrons de tests).

1.2.2.2 Logiciels utilisés

Le domaine des « logiciels utilisés » inclut les logiciels de compilation, leurs bibliothèques associées, le système d'opération utilisé et les autres bibliothèques ou logiciels dont le logiciel créé dépend.

Il est d'usage de considérer les « logiciels utilisés » comme corrects. Advenant une non-conformité dans un de ceux-ci, il incombe généralement au fournisseur d'apporter les correctifs nécessaires selon les dispositions de la licence du logiciel.

1.2.2.3 Matériel

Les tests du domaine « matériel » sont classés en deux groupes (performance et stabilité) qui couvrent, de façon non exhaustive, les éléments suivants :

1. Performance;
2. Stockage;
3. Vitesse de calcul;
4. Vitesse de débit (ex. : kilobits/sec);
5. Temps de réponse (ex. : temps d'exécution de 1000 requêtes à une base de données);
6. Stabilité;
7. Temps moyen avant bris matériel (MTTF : *Mean Time To Failure*);
8. Fonctionnement incorrect du matériel.

1.2.2.4 Humain

Les tests du domaine « humain » sont reliés à quatre types d'erreurs :

1. Besoin exprimé incorrectement par le client dans la spécification du logiciel;
2. Compréhension incorrecte de la spécification du logiciel par le programmeur;
3. Implémentation incorrecte de la spécification du logiciel;
4. Utilisation incorrecte du matériel ou des logiciels utilisés.

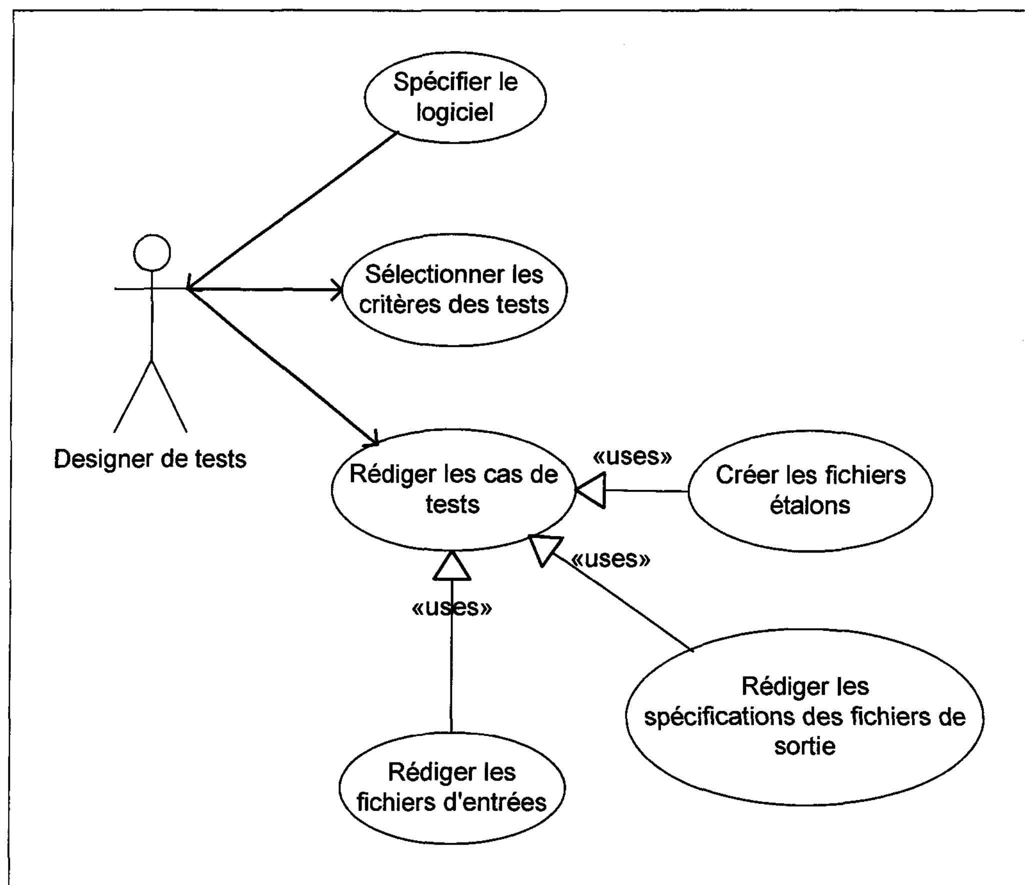
Notons que les tests de matériel qui relèvent d'un choix humain — utilisation du mauvais matériel ou utilisation incorrecte du matériel — se placent dans le domaine humain et non dans le domaine matériel. Cependant, dans tous les cas, la validation de ce domaine requiert l'action d'un humain.

1.2.3 Test a priori

Dans le cas d'un test a priori, le résultat obtenu doit être comparé avec un étalon pour le test concerné. Cette comparaison donne le verdict du test. Si le résultat est identique à l'étalon, le test est conforme. Il est non conforme dans le cas contraire.

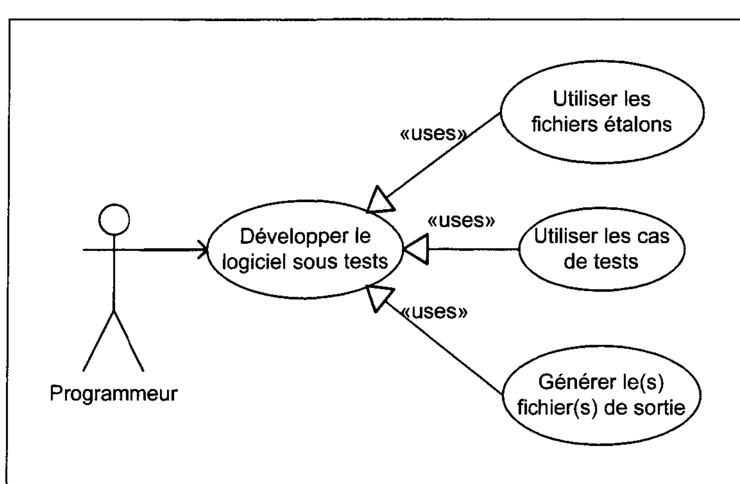
Avant la création d'un test a priori, le client et l'analyste précisent les spécifications du logiciel. Dans le contexte présent, le terme analyste désigne le représentant de l'équipe de développement qui aide le client à exprimer ses besoins. À l'aide des spécifications du logiciel, l'analyste précise les critères de sélection de tests de logiciel. Ceux-ci pourront changer selon l'évolution du projet. Les figures 10 et 11 montrent les étapes la conception et la construction d'un test a priori.

Figure 10 : Conception d'un test a priori



Durant l'étape « conception », le concepteur de tests utilise les spécifications du logiciel et les critères de sélection des tests pour générer les intrants/extrants des tests ainsi que les comportements attendus et les domaines d'utilisation. Dans ce cas, un extrant sera l'étalon que le programmeur utilisera pour valider son implémentation. Les cas de tests peuvent inclure des parties des spécifications du logiciel ou tout autre information permettant au programmeur d'être efficace et de comprendre l'implémentation de ceux-ci.

Figure 11 : Construction d'un test a priori

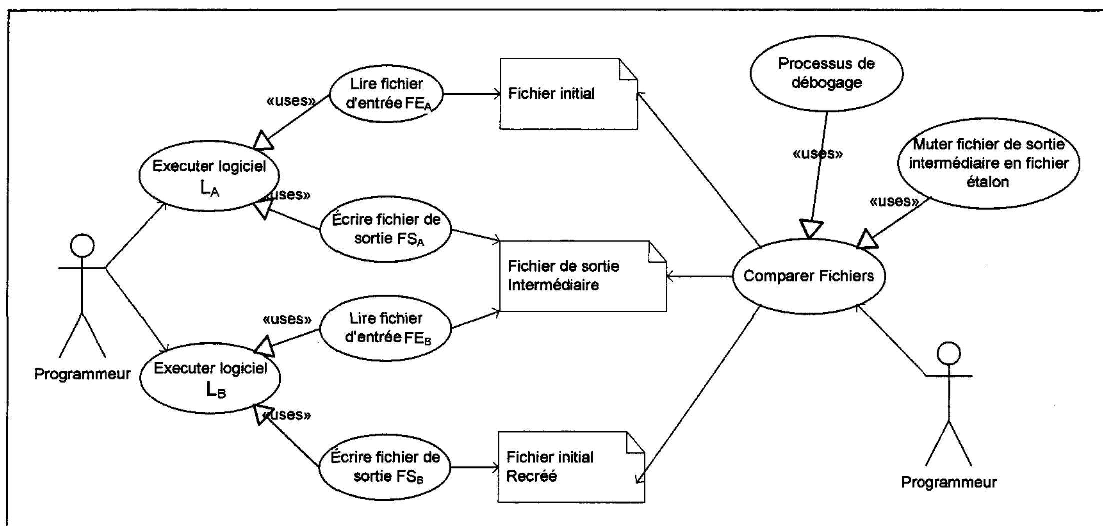


Durant l'étape de construction d'un test a priori, le programmeur utilise les fichiers étalons et les documents des cas de tests pour implémenter le code informatique qui générera des fichiers de sortie qui seront les « alter ego » des fichiers étalons. Lorsque le test a priori est construit, il est inséré, le cas échéant, dans les tests de régression (section 1.2.5 : Test de régression). L'exécution des tests de logiciel permet de fournir une image à jour de l'évolution du logiciel en comparant les fonctionnalités testées et celles à implémenter.

1.2.3.1 Cas particulier

Dans certains cas, il est impossible au concepteur de tests de fabriquer un étalon. Par exemple : il est demandé de faire une fonction qui crypte un fichier selon un algorithme propriétaire. L'étalon n'est donc pas accessible avant l'implémentation de la fonction. Toutefois, le concepteur de tests peut contourner cette difficulté en donnant au programmeur la procédure pour générer la réponse attendue. La figure 12 montre ce cas particulier.

Figure 12 : Méthode pour générer un fichier étalon



Cette méthode utilise un logiciel L_A pour lire un fichier d'entrée FE_A et générer un fichier de sortie FS_A, qui est en fait un fichier de sortie intermédiaire. Ici, le FE_A est le « Fichier initial » du test. Un second logiciel, ici L_B, lit le fichier d'entrée FE_B et génère un fichier de sortie FS_B. Ce dernier est alors le « Fichier initial recréé ».

Pour terminer, le programmeur compare « Fichier initial » et le « Fichier initial recréé ». Si la comparaison est correcte, le « Fichier de sortie intermédiaire » sera muté en fichier étalon. Le fichier étalon sera utilisé lors des tests de régression. Si la comparaison est incorrecte, au moins une des deux implémentations est fautive.

Si les deux implémentations étaient fautives, mais donnaient quand même un résultat correct, l'erreur serait probablement détectée lors d'utilisation ultérieure de la fonction par d'autres composants. Il n'existe pas de moyen de détecter autrement ce type d'erreur. À remarquer que les actions des logiciels L_A et L_B sont l'inverse l'une de l'autre.

1.2.4 Test a posteriori

Un test a posteriori est réalisé après la fabrication d'une version pleinement fonctionnelle. Il est effectué par le client ou par un groupe de bêta-testeurs. Cette méthode est très utilisée dans les milieux *Open Source* (section 1.1.2.5).

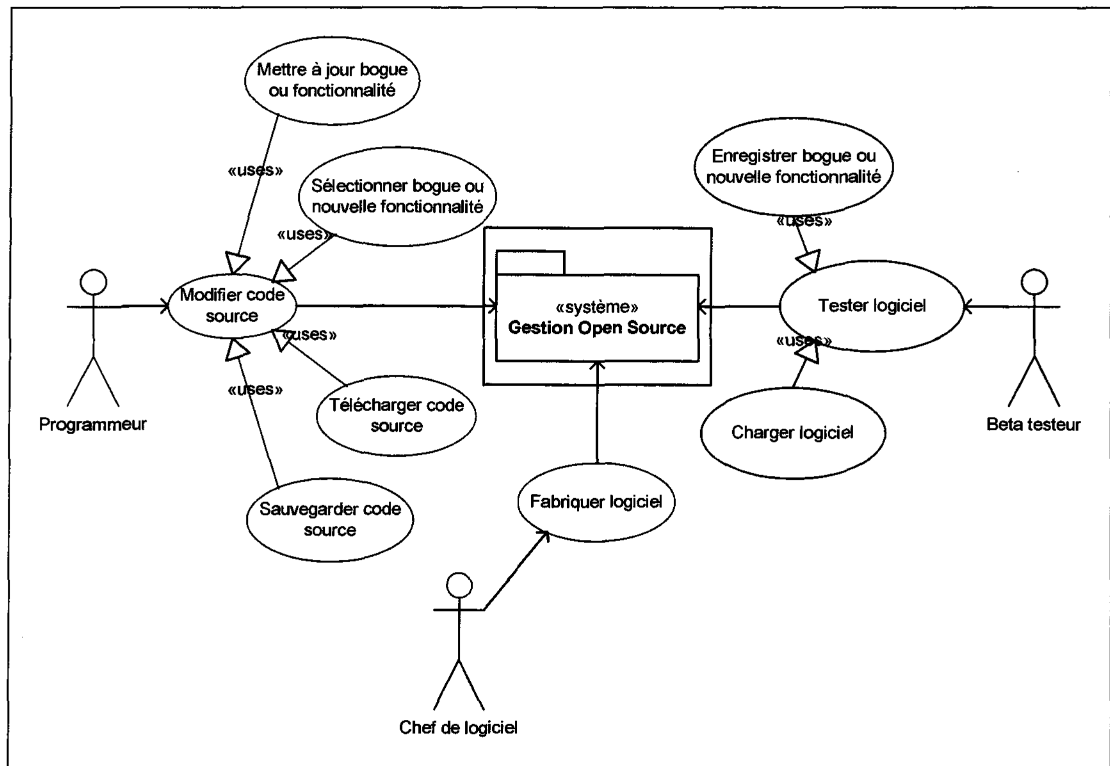
Il n'est pas rare de voir des logiciels avec un groupe de bêta-testeurs de l'ordre de plusieurs centaines voire quelques milliers [36], sinon plus. C'est précisément dans ce facteur numérique que réside la force des tests a posteriori de la méthode *Open Source* : des bêta-testeurs par milliers versus quelques testeurs, ou guère plus, pour les entreprises fabriquant du logiciel.

Le test a posteriori est basé sur une boucle de rétroaction entre le bêta-testeur et le programmeur. Cependant, selon la méthode de développement *Open Source*, la fabrication du logiciel est la responsabilité du chef de logiciel et c'est celui-ci qui regroupera plusieurs modifications en une nouvelle version du logiciel. Cette action de regroupement cause un certain retard entre l'enregistrement et la réparation des bogues ou entre l'inscription de nouvelles fonctionnalités et leurs réalisations dans une nouvelle version du logiciel. Aux fins de simplicité dans la figure 13, il est supposé que le logiciel fabriqué est placé sur le même serveur que le code source. En pratique, ceci est rarement le cas. Toutefois, cette simplification ne modifie en rien le fonctionnement de la boucle de rétroaction.

Le modèle ci-dessous montre l'utilisation d'un système de gestion *Open Source.* Celui-ci est au cœur de la méthode de test a posteriori. Un des systèmes les plus connus est le site de *Source Forge* [37] qui contient plus de 100,000 projets dans sa banque de données.

La figure 13 montre un modèle simplifié de la méthode de test a posteriori.

Figure 13 : Test a posteriori (boucle rétroaction)



Lorsque le bêta-testeur veut tester le logiciel, il doit tout d'abord charger celui-ci. Puis, durant l'utilisation/test du logiciel, le bêta-testeur prend note des bogues rencontrés ou des nouvelles fonctionnalités qui pourraient être intéressantes. Ensuite, il enregistre chacun de ceux-ci dans le système de gestion.

De son côté, le programmeur parcourt la liste de tous les bogues et des suggestions de nouvelles fonctionnalités et y sélectionne le bogue ou la fonctionnalité qu'il désire implémenter. Ensuite, il télécharge le code source correspondant, modifie celui-ci, puis le sauvegarde. Quant au chef de projet, il suit l'évolution des bogues corrigés ou des fonctionnalités implémentées et, à son jugement, fabrique une nouvelle version et la rend disponible aux bêta-testeurs.

C'est à ce moment que la boucle de rétroaction est complétée. Le bêta-testeur amorcera un nouveau cycle en testant la nouvelle version du nouveau

logiciel et en rapportant les bogues et les nouvelles fonctionnalités. À son tour, le programmeur sélectionnera un bogue ou une nouvelle fonctionnalité et modifiera le code source correspondant pour ensuite le sauvegarder, créant ainsi une rétroaction.

1.2.5 Test de régression

Un test de régression est défini [13] par la répétition de tests pour s'assurer que les modifications apportées n'ont pas introduit de fautes dans un logiciel préalablement correct. Les exécutions des tests de régressions se répartissent en deux approches principales [38] :

- Retester tout : ré-exécuter tous les tests disponibles (demande une grande consommation de ressources et de temps);
- Retester une sélection : ré-exécuter un sous-ensemble de tous les tests disponibles (difficulté de choisir les bons tests, ce qui pourrait permettre à une faute nouvellement introduite de passer inaperçue).

Il existe principalement deux critères pour la sélection des tests de régression dans l'approche d'exécution sélective :

- Couverture : cherche à satisfaire un critère de couverture de code. Par exemple pour la couverture de code, 80 % des blocs de code doivent avoir au moins un test qui les parcourent;
- Minimisation : comme le critère de couverture mais avec une contrainte supplémentaire visant à minimiser l'effort de test. Par exemple : choisir un seul test à exécuter pour chaque composant modifié.

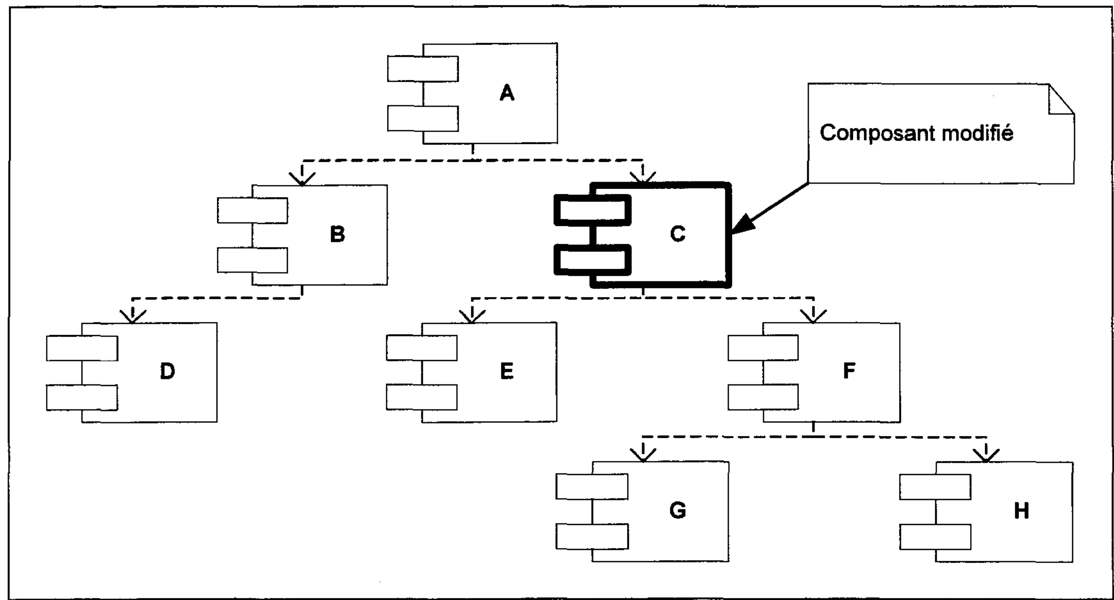
Une notion importante dans les tests d'unités logiciel est le chaînage des composants[6]. La connaissance de ce chaînage de composants permet de cibler les tests de façon plus précise par la connaissance des plus proches voisins. Un plus proche voisin est défini comme étant un composant qui fait appel directement à une fonction d'un autre composant.

[6] Un composant peut avoir quelques centaines de lignes de code jusqu'à plusieurs milliers.

La figure 14 montre un exemple de ce chaînage de composants. Le composant **A** utilise des fonctions des composants **B** et **C**. Le composant **B** utilise des fonctions du composant **D** et ainsi de suite. Dans cette même figure, les composants **G**, **H** et **B** sont les 2ième voisins du composant **C**.

Posons que des modifications ont lieu dans le composant **C**, ce qui est représenté par le pourtour du composant en caractère gras.

Figure 14 : Structure des unités logiciel



Pour représenter la correspondance entre un composant et ses tests associés, ceux-ci sont représentés par des lignes obliques à l'intérieur du composant en question. Si le composant est totalement hachuré, c'est que tous les tests de ce composant sont sélectionnés pour être exécutés. Si le composant est partiellement hachuré, alors seulement une partie des tests a été sélectionnée.

Dans l'exemple précédent, en omettant le cas trivial de ne refaire aucun test, il existe cinq façons de sélectionner les tests à exécuter. Ceci est présenté au tableau 1 :

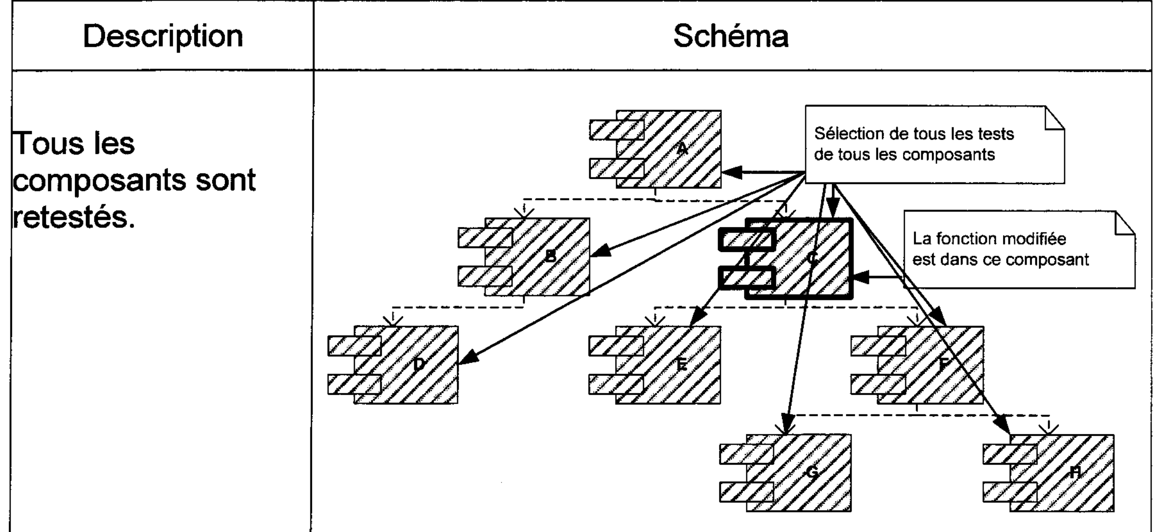
Tableau 1 : Structure incrémentale des tests de régression

Description	Schéma
Seulement les tests reliés à la fonction modifiée sont ré-exécutés.	A, B, C, D, E, F, G, H — Sélection partielle des tests — La fonction modifiée est dans ce composant
Tout le composant est retesté.	A, B, C, D, E, F, G, H — Sélection de tous les tests du composant — La fonction modifiée est dans ce composant

Tableau 1 : Structure incrémentale des tests de régression(suite)

Description	Schéma
Tout le composant modifié est retesté ainsi que les fonctions qui appellent ou qui sont appelées par le composant modifié. Ceci permet de valider les interfaces entre les composants. Note : la principale difficulté est de coordonner les tests des fonctions appelantes/appelées et le composant modifié.	
Tout le composant est retesté ainsi que les composants qui sont les « plus proches voisins ». Note : Probablement la plus simple à implémenter dans un système de version concourante.	

Tableau 1 : Structure incrémentale des tests de régression (suite)

Description	Schéma
Tous les composants sont retestés.	

Il est à noter que dans le cas où une petite entreprise a automatisé des tests de régression, il est souvent plus simple et rapide de faire un grand nombre de tests plutôt que de prendre le temps de sélectionner manuellement les tests les plus pertinents. Il est important de garder en tête que l'automatisation des tests de régression n'est pas une panacée et que certains de ces tests sont très longs à automatiser ou demandent beaucoup de maintenance. De ce fait, ils engendrent un coût monétaire important pour une faible rentabilité.

1.3 Méthodes de tests

Il existe trois méthodes définissant les façons de produire les tests. La première méthode de tests s'appelle la boîte blanche. C'est lorsque le programmeur a accès au code source. La seconde méthode est celle dite de la boîte noire. Dans ce cas, le programmeur n'a pas accès au code source. La troisième méthode est celle de la boîte grise. Celle-ci est l'intermédiaire entre la méthode de la boîte blanche et celle de la boîte noire.

1.3.1 Boîte blanche

La méthode de test de type boîte blanche est celle où le programmeur accède directement au code. La locution « boîte blanche » vient du fait que, lors des tests, le fonctionnement interne du composant est connu [34]. C'est en connaissant le code informatique à l'intérieur des composants que le programmeur peut valider le cheminement logique de l'exécution du logiciel et, le cas échéant, modifier le test pour avoir le comportement approprié.

1.3.2 Boîte noire

La méthode de test de type boîte noire, aussi nommée test fonctionnel [34], est celle où le programmeur ignore les mécanismes internes d'un système ou d'un composant testé et porte son attention exclusivement sur les sorties générées en fonction d'entrées sélectionnées et des conditions d'exécution.

1.3.3 Boîte grise

La méthode de test de type boîte grise est une combinaison des méthodes boîte blanche et boîte noire. Les tests sont effectués en fonction des interfaces extérieures au composant, mais le choix des tests est effectué en connaissant les mécanismes internes des composants et la façon dont ils interagissent entre eux. Usuellement, cette méthode est utilisée comme phase de transition entre les tests en boîte blanche et ceux en boîte noire.

2

MÉTHODES USUELLES DE TESTS

Selon Edsger Wybe Dijkstra [39] (traduction) :

Les tests de logiciel peuvent être utilisés pour montrer la présence de bogues, mais jamais pour montrer leur absence.

Il est donc impossible de montrer que le logiciel est sans bogue, sauf exception d'un logiciel trivial. Il est cependant possible de montrer que le logiciel fonctionne pour un certain nombre de cas de tests. Les tests sont utilisés pour mesurer la concordance entre les spécifications et ce que fait le logiciel.

De nombreux sites Internet, Foires Aux Questions (FAQ) ou lettres électroniques [40;41;42;43] sont dédiés aux professionnels des tests logiciels. Au moins un site [40] liste plus de 350 outils qui peuvent être utilisés pour l'exécution de tests. Un survol de plus de 150 de ces outils a permis de trouver des outils gratuits pour le développeur jusqu'à des systèmes corporatifs valant plus de 150,000 $. De même, il s'y retrouve aussi des trucs, des méthodes, des techniques, des théories, des définitions, des démonstrations mathématiques ainsi que des listes d'incidents causés par des bogues informatiques.

Les premières sections de ce chapitre décrivent plusieurs méthodes de tests ou de génération de données pour les tests de logiciel. Suivra une courte section sur l'utilisation de la gestion des erreurs de logiciel pour aider à documenter les bogues. Des remarques concernant les tests d'interfaces graphiques et quelques logiciels de tests terminent ce chapitre.

2.1 Conditions aux frontières

La méthode de test des conditions aux frontières s'applique lorsqu'au moins un des paramètres d'une fonction est numérique ou est un objet/structure ayant des champs numériques. L'idée est de borner le domaine des nombres valides par des frontières inférieures et supérieures.

Le programmeur a comme tâche de construire un logiciel qui travaille avec des nombres valides et qui gère de façon prévisible les nombres invalides. Les spécifications du logiciel précisent si la frontière fait partie des nombres valides ou des nombres invalides. L'intérêt de cette méthode est de permettre au programmeur de valider les données d'entrée et de corroborer si l'emploi des opérateurs de comparaison (<, <=, ==, etc.) est correct.

Par définition, la locution « domaine des valeurs d'entrée » représente tous les nombres valides qui satisfont une spécification du logiciel. Le tableau 2 montre quelques exemples simples de domaine des valeurs d'entrée :

Tableau 2 : Exemples de domaine de valeurs d'entrée

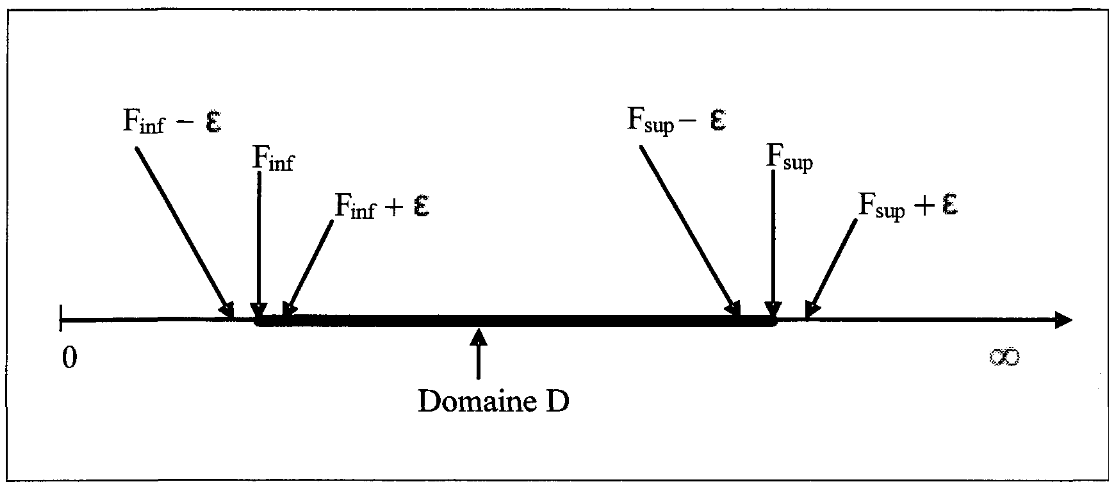
Domaine des valeurs d'entrée[7]	Description
N = [0, ∞[	Nombres positifs, de 0 à l'infini.
M = [0, 9] et [20, 30]	Nombres compris de 0 à 9 et ceux de 20 à 30 inclusivement.
P =]-∞, ∞[	L'ensemble des nombres allant de moins l'infini à plus l'infini.
Q = [6,10]	Les nombres compris entre 6 et 10 inclusivement.
S = [32, 127]	Les nombres entre 32 et 127 inclusivement.

[7] Les domaines des valeurs d'entrées peuvent être des sous-ensembles des entiers ou des réels; le contexte des spécifications de logiciel indiquera au programmeur le choix à effectuer.

Ces nombres se rapportent à des articles dans la spécification des besoins. Par exemple, un numéro de client, la grandeur d'un champ texte, une plage salariale, etc. Dans le tableau précédent, pour plus de précision, il aurait été nécessaire de mentionner si le logiciel était en 16, 32, ou 64 bits et de remplacer « infini » par la valeur correspondante. En langage C, ces informations sont dans le fichier d'en-tête « `limits.h` ». Aussi, la spécification du logiciel pourrait mentionner que le domaine S, dans le tableau 2, représente en fait les caractères imprimables, sans accent, d'une table ASCII.

Un aspect important dans cette méthode est de travailler aussi près que possible de la frontière, mais de part et d'autre de celle-ci. La figure 15 montre les éléments importants de cette méthode.

Figure 15 : Conditions aux frontières



Le domaine D est borné par les frontières inférieure (F_{inf}) et supérieure (F_{sup}). Les éléments les plus près de ces frontières sont ceux qui sont à une distance de $\pm\varepsilon$ de celles-ci sachant que ε tend vers zéro. Cependant, en informatique, de par la nature discrète des nombres, ε possède une valeur finie qui est différente de zéro et qui dépend du type de données. Par exemple, dans le cas du type entier, la valeur de ε est égale à 1.

Voici un exemple simple où une fonction doit valider que le nombre reçu en paramètre soit plus grand ou égal à 0 et plus petit ou égal à 12. Dans ce cas, le domaine s'écrit D = [0, 12], avec les valeurs frontières : 0 et 12.

Une implémentation en langage C, d'une fonction quelconque `foo` qui utilise des valeurs de ce domaine, pourrait être comme suit :

Exemple 1 : Conditions aux frontières

```
bool foo(int iNb)
{
   if ((0 <= iNb)
       && (iNb<= 12) )
      return true;
   else
      return false;
}
```

Puisque les frontières sont à 0 et à 12, il y aura six éléments à tester, trois pour la frontière inférieure et trois pour la frontière supérieure. Pour chacun de ces six éléments, les résultats attendus pour les valeurs retournées sont les suivantes.

Tableau 3 : Conditions aux frontières

Frontière	Paramètre (int iNb)	Résultat attendu
Inférieure	-1	Faux
	0	Vrai
	1	Vrai
Supérieure	11	Vrai
	12	Vrai
	13	Faux

L'intérêt de cette méthode est qu'elle permet de capter des erreurs d'utilisation des opérateurs de comparaison (<, <=, ==, etc.). Cet exemple simple

amène une question très importante : « Comment traiter les capacités limites de nombres ? ».

Cumming [44] rapporte un algorithme de la compagnie Motorola pour gérer les dépassements de capacité des nombres entiers. D'autres méthodes plus simples existent, mais elles sont cependant plus lentes. Voici un exemple de code, en langage C, qui implémente une telle méthode et les résultats de l'exécution de chaque ligne de code.

Tableau 4 : Dépassement de capacité

Ligne	Implémentation	Résultat
1	`long x = LONG_MAX`[8]`;` `long y = 0;`	Déclaration et initialisation des variables (LONG_MAX = 2147483647L)
2	`y = x + x;`	y = -2 (dépassement de la valeur)
3	`y = x;`	y = 2147483647

Dans le tableau précédent, le résultat **y** (ligne n°2) est plus petit que la valeur **x** dont il est issu, ce qui est un symptôme du dépassement de capacité. Une règle simple est déduite : il y a dépassement de capacité si le résultat de l'addition de deux nombres positifs est plus petit que chacun des deux nombres desquels il est issu. Si les deux nombres sont négatifs, il y aura dépassement de capacité dans le cas de l'addition si le résultat est plus grand que chacun des deux nombres desquels il est issu. Il ne peut y avoir de dépassement de capacité lors de l'addition d'un nombre positif et d'un nombre négatif. Si une vérification de dépassement de capacité est effectuée à chaque opération, le coût en temps pour effectuer cette opération s'additionne au temps d'exécution du logiciel et le ralentit d'autant. La Compagnie IBM [45] présente une excellente ressource pour le

[8] LONG_MAX provient du fichier d'en-tête `limits.h`, en langage **C**.

traitement des nombres décimaux, des nombres à dimension fixe et ceux à virgule flottante.

Il existe un autre cas de dépassement de capacité. Il s'agit de la transformation d'une chaîne de caractère en nombre. Dans ce cas, l'utilisation des fonctions d'analyse de chaîne de caractère permet de bien gérer le comportement. Un exemple type de ces fonctions est celui du comportement de la fonction `strtol(...)` du langage C qui place son code d'erreur dans la variable globale `errno`.

Pour terminer cette section, voici un exemple complet qui illustre l'utilisation de la méthode de conditions aux frontières à partir de la spécification du logiciel jusqu'à la rédaction des tests et des valeurs attendues. Une fonction validera un mot de passe de 6 à 10 caractères. Ce mot de passe pourra être formé de toutes combinaisons de lettres minuscules et majuscules, sans accent, et de chiffres.

Tableau 5 : Spécifications de la fonction de validation du mot de passe

Spécifications[9]	Valeur/Description
Lettres valides	Les lettres (minuscules et majuscules) sans accent et les chiffres (de 0 à 9)
Nombre minimal de caractères	6 (frontière inférieure)
Nombre maximal de caractères	10 (frontière supérieure)
Lettres minuscules	a à z
Lettres majuscules	A à Z
Chiffres	0 à 9
Règle de composition	Peut être composée entièrement de chiffres ou de lettres ou d'un mélange de ceux-ci.

Pour simplifier cet exemple, il est supposé qu'il existe une fonction qui vérifie si le caractère est valide ou non selon les spécifications. Il restera alors

[9] Il est implicite que le nombre de caractères du mot de passe est un nombre entier.

seulement à valider si l'implémentation de la spécification de la longueur du mot de passe est conforme.

Dans cet exemple, il existe deux frontières pour la longueur : la frontière inférieure ($F_{inf} = 6$) et la frontière supérieure ($F_{sup} = 10$). Alors, le domaine des valeurs d'entrée est D = [6, 10]. Le domaine D contient toutes les valeurs dans l'intervalle donné.

Pour tester les conditions aux frontières, il importe de valider trois cas pour chaque frontière :

1. À l'extérieur de la frontière;
2. Sur la frontière;
3. À l'intérieur de la frontière.

La figure 16 montre ces conditions.

Figure 16 : Conditions aux frontières (D = [6, 10])

$F_{inf} - 1$
F_{inf}
$F_{inf} + 1$
$F_{sup} - 1$
F_{sup}
$F_{sup} + 1$
0
5 6 7
9 10 11
∞
Domaine D

Il y a donc six tests à effectuer, soit des chaînes de 5, 6, 7, 9, 10 et 11 caractères. Le tableau ci-après montre la longueur de la chaîne, la valeur de la chaîne de caractères et le résultat attendu[10].

[10] Puisque le nombre de caractères est « petit », il peut être aussi acceptable de parcourir tous les éléments du domaine **D**.

Tableau 6 : Chaîne test de conditions aux frontières

Frontière	Longueur de la chaîne	Chaîne test	Résultat attendu
$F_{inf} - 1$	5	FooBa	erreur
F_{inf}	6	FooBar	correct
$F_{inf} + 1$	7	FooBarF	correct
$F_{sup} - 1$	9	FooBarFoo	correct
F_{sup}	10	FooBarFooB	correct
$F_{sup} + 1$	11	FooBarFooBa	erreur

Dans la plupart des cas, nombre entier, nombre à point fixe, caractère ou chaîne de caractères, il est assez simple de trouver les valeurs frontières. Cependant, dans le cas des nombres réels, la variabilité sur le bit le moins significatif peut causer des difficultés.

2.2 Force brute

La méthode de tests par force brute consiste à utiliser la puissance de l'ordinateur pour générer tous les cas de tests et les exécuter. L'exemple type est le suivant : une fonction qui additionne deux entiers signés de 32 bits.

Exemple 2 : Additionneur

```
int somme(int num1, int num2)
{
   return (num1 + num2);
}
```

Dans l'exemple présenté plus haut, le nombre de cas générés pour satisfaire le cas de force brute est N x N, avec $N = 2^{32}$. Le nombre de cas à tester est donc de 2^{64}, ce qui est un très grand nombre. Posant qu'un ordinateur exécute 10^9 opérations par seconde, cela donnerait comme temps requis environ 585 années. Ce temps de tests n'est pas humainement raisonnable.

2.3 Injection volontaire de fautes logicielles

L'injection volontaire de fautes logicielles est basée sur une méthode bien connue dans le domaine du matériel électronique. Elle consiste à fournir au matériel des valeurs non attendues et à observer le comportement. Dans le cas d'un logiciel, l'injection volontaire de fautes logicielles fournit au logiciel des valeurs non conformes. En ce sens, elle peut être considérée comme un sous-ensemble de la méthode de conditions aux frontières (section 2.1 : Conditions aux frontières). Cependant, deux variations de cette méthode existent, soit l'injection de fautes dynamiques et l'injection statique. Cette dernière est souvent appelée mutation de code.

2.3.1 Injection dynamique

Bieman, Dreilinger et Lijun [46] mentionnent que l'injection dynamique est souvent utilisée pour simuler des erreurs de matériel. Celles-ci sont créées en modifiant ou en injectant des fautes dans les bits de mémoire ou dans les registres.

2.3.2 Injection statique (mutation de code)

La méthode de mutation de code consiste à modifier une condition booléenne du code sous tests et à observer le comportement du logiciel. Bieman et al.[46] décrivent le processus de fonctionnement de la méthode. Cette méthode de tests est inusitée, principalement dû au fait qu'elle consiste à insérer volontairement des erreurs de logique dans le code source pour observer le comportement du logiciel et des tests reliés. D'autre part, les auteurs mentionnent que le nombre de logiciels mutants nécessaires pour tester les injections est très grand. S'il y a N variables, un logiciel pourrait avoir N^2 versions de logiciels mutants, ce qui demanderait des ressources massives pour recompiler et exécuter toutes ces versions.

L'exemple 3 donne une illustration de l'application de cette méthode. Le code informatique vérifie que l'eau ne bout pas à une température inférieure à 100° C à une pression de 760mm de mercure [47].

Exemple 3 : Eau ne bout pas

```
if (T_Celcius < 100)
{
   printf(«L'eau ne bout pas.»);
}
else
{
   printf(«L'eau bout.»);
}
```

Dans cet exemple, le domaine des valeurs d'entrée est l'ensemble des nombres entiers positifs ou négatifs, incluant zéro. La frontière des données valides et invalides est à la température T_Celcius < 100. Donc, la validation sera autour de ce point. Voici les tests à la frontière et les résultats attendus.

Tableau 7 : Résultats des tests (température de l'eau)

Frontière	Valeur de la température à la frontière	Affichage	Valeur étalon	Verdict
T_Celcius – 1	99	L'eau ne bout pas	L'eau ne bout pas	correct
T_Celcius	100	L'eau bout	L'eau bout	correct
T_Celcius + 1	101	L'eau bout	L'eau bout	correct

Dans cet exemple, un test d'erreur volontaire sur l'opérateur de comparaison < pourrait être de le remplacer par <= ou par =. Ce remplacement devrait apporter au moins une modification aux résultats de l'ensemble des tests. Dans le cas où il n'y a pas de modification générée, il est alors certain que la condition n'a pas été validée correctement. Voici les résultats qui seraient obtenus si l'opérateur < était remplacé par <=.

Tableau 8 : Résultats tests (température de l'eau, après mutation de code)

Frontière	Valeur de la température à la frontière	Affichage	Valeur étalon	Verdict
T_Celcius – 1	99	L'eau ne bout pas	L'eau ne bout pas	correct
T_Celcius	100	L'eau ne bout pas	L'eau bout	erreur
T_Celcius + 1	101	L'eau bout	L'eau bout	correct

La différence dans les verdicts était ce qui était attendu. L'exécution de cette méthode de tests permet de valider que les données utilisées sont effectivement sur la frontière des opérateurs de comparaison.

Dans l'exemple présenté plus haut, si les données de tests avaient été prises de façon quelconque (par exemple : 50 et 150), les tests effectués par le programmeur auraient été valides. Cependant, après l'injection volontaire de la faute, il n'y aurait eu aucune variation dans les sorties, indiquant par le fait même un symptôme à investiguer.

2.4 Génération aléatoire des tests

La locution « génération aléatoire des tests », ou test des singes, est une traduction libre de l'expression anglaise *monkey testing* qui est elle-même une traduction populaire de la locution : *random testing*[11]. L'emploi du mot singe fait allusion à l'expression connue (traduction) :

> *Six singes tapant sur six claviers de dactylo de façon aléatoire pendant un million d'années vont recréer tous les écrits d'Isaac Asimov.*

Cependant, certains préfèrent le terme technique : tests stochastiques.

[11] Ici, il faut différencier *random testing* qui s'applique aux transitions de fonctions et le *random data generation* (section 2.5 : Génération aléatoire des données) qui s'applique à générer des données pour couvrir tout l'arbre du flot de contrôle.

Nyman [48] donne une bonne description de singes savants et singes ignorants de même que les domaines d'utilisation de chacun. Notons simplement qu'un singe savant est un automate qui connaît la machine d'état d'une application et qui traverse celle-ci aléatoirement dans les transitions permises. Un singe ignorant parcourt le même diagramme d'état, mais il utilise aléatoirement tous les états possibles, essayant, par le fait même, des transitions illégales. Un exemple de cette utilisation : les tests d'interface graphique où le nombre d'états est grand et la complexité élevée.

Posons une matrice de transition M, de 5x5, pour une machine d'état déterminée. Les colonnes représentent l'état actuel et la ligne l'état désiré (ou futur). La valeur qui est dans l'intersection indique si la transition est permise (valeur = 1) ou non permise (valeur = 0). Dans le tableau 9, l'état de départ est situé dans le haut du tableau et l'état d'arrivée est sur le côté gauche. Par exemple : la transition A→B est permise (valeur =1) alors que la transition B→A est non permise (valeur =0).

Tableau 9 : Matrice de transition

État	A	B	C	D	E
A	0	0	0	0	1
B	1	0	1	1	1
C	1	1	0	0	0
D	0	1	1	0	0
E	0	0	0	1	1

Dans l'exemple ci-dessus, les tests aléatoires peuvent s'appliquer exclusivement sur les 11 transitions permises (singe savant) ou sur toutes les 25 transitions (singe ignorant).

Cette méthode de tests s'applique bien à des logiciels qui sont constitués d'un assemblage de composants interagissant entre eux (par exemple : les interfaces graphiques). En raison du grand nombre de transitions possibles, un

pourcentage de transition doit être déterminé et atteint pour que les tests soient considérés comme valides.

Par exemple : dans un logiciel qui aurait 50 000 états, le nombre de transitions possibles entre chacun de ces états est alors de $2.5x10^9$ transitions. Supposant qu'un ordinateur soit capable de faire 1 000 tests de transitions par seconde, incluant l'écriture de résultats dans des fichiers, il faudrait alors environ 28.9 jours pour tester toutes ces transitions. Ceci ne teste que les transitions, et non le contenu des fonctions appelées.

2.5 Génération aléatoire des données

Cette méthode consiste à générer de manière pseudo aléatoire des données qui respectent les domaines des valeurs d'entrée des paramètres utilisés. Cette méthode s'applique dans le cas où un nombre restreint de données est suffisant ou lorsqu'il est inadéquat d'utiliser la force brute. La fonction pseudo aléatoire doit générer les données avec une équiprobabilité sur tout le domaine des valeurs d'entrée.

Parmi les méthodes logicielles de génération des données, la méthode aléatoire donne les moins bons résultats sur le plan de la couverture de code. Le morceau de code suivant montre ce fait.

Exemple 4 : Génération aléatoire des données (cas problème)

```
int foobar(int a, int b)
{
   if (a == b)
      imprime(a);
   else
      imprime(b);
}
```

La probabilité d'activer `imprime(a)` est de 1/n, avec n le nombre d'éléments dans le domaine des nombres entiers (`int`).

2.6 Génération automatique des données

Le Graal de l'assurance-qualité est la génération automatique des données de tests. La prémisse de base est que le temps machine est moins coûteux que le temps humain. Donc, si un logiciel pouvait générer les données qui permettraient de tester les cas de branchement (`if`, `while`, `switch`, etc.) de façon adéquate, le coût de développement d'un logiciel serait diminué.

La génération automatique des données de tests consiste à utiliser soit le langage source (par instrumentation avant compilation), soit certains codes binaires pour générer les cas de tests. Comme il sera présenté un peu plus loin, seulement certains codes binaires peuvent être utilisés pour tirer des tests de ceux-ci. La génération des cas de tests est basée sur des algorithmes qui tentent d'explorer tous les branchements logiques d'un logiciel. Edvardsson [49] sépare cette méthode en trois phases distinctes : un analyseur de logiciel, un sélecteur de chemin et un générateur de données.

L'analyseur de logiciel génère un graphe de flot de contrôle*. Les nœuds sont les blocs de code avec un seul début et une seule fin, sans branchement ni arrêt, sauf à la fin du bloc. Les arêtes indiquent un prédicat pour passer d'un nœud à un autre. Dans le cas où le prédicat est vide, il est toujours vrai, par exemple, le cas de deux blocs de code séquentiels. Dans chaque graphe, il y a un seul nœud d'entrée et un seul nœud de sortie. À n'importe quel moment, aucun nœud ne peut avoir deux arêtes dont les conditions de prédicat sont vraies. Dans le cas contraire, le logiciel serait alors non déterministe.

Un outil de sélection de chemin détermine quels seront les chemins complets[12] ou les chemins partiels[13] qui seront soumis aux tests. Le générateur de

---

[12] Chemin complet : un chemin qui débute au nœud d'entrée et se termine au noeud de sortie.
[13] Chemin partiel : un chemin qui n'est pas un chemin complet.

données tente d'analyser le domaine des valeurs d'entrée de chaque chemin reçu qui permettrait de valider tous les branchements de ces chemins.

Gotlieb, Botella et Rueher [50] ont développé un algorithme de génération automatique des données dont les tests ont été effectués sur un sous-ensemble non trivial du langage C. L'algorithme est conçu pour être exécuté sur le code source. Les éléments non structurés du langage C, tel que le *goto,* ont été exclus. De même, selon les auteurs, l'arithmétique des pointeurs, l'allocation dynamique des structures, les pointeurs à des fonctions et le *type casting* impliquent des problèmes difficiles à résoudre et sont donc exclus.

Lors d'une comparaison expérimentale, Gotlieb et al. ont mesuré la performance de leur algorithme de génération de données « InKa » contre un logiciel utilisant une méthode de génération aléatoire « Random ». Cette expérimentation consiste à générer des ensembles de données de tests pour quatre logiciels et à mesurer la couverture de code dans ceux-ci. Les résultats sont intéressants, comme le montre le tableau 10.

Tableau 10 : Comparaison couverture de code

Logiciels testés	LOC	Blocs	Test data	Logiciels de génération des données de tests		
				Testgen[14]	Random	InKa[15]
bsearch	21	10	$>10^{50}$	100%	100%	100%
sample	33	14	$>10^{100}$	100%	93%	100%
trityp	40	22	$>10^{10}$	100%	86%	100%
ardeta03	157	38	$>10^{60}$	N/D	74%	100%

[14] 50 minutes par bloc, sur un PC (60Mhz).
[15] 10 secondes par bloc, sur un Sun Sparc 5 (300Mhz) sous Solaris 2.5.

Dans la première colonne se trouvent les noms des logiciels dont les codes source ont été utilisés pour être testés. De ceux-ci, seul le logiciel « ardeta03 » est de fabrication commerciale, les autres ayant été développés sur des cas académiques. La deuxième colonne représente le nombre de *Line Of Code* (lignes de codes) de chaque code source testé. La troisième colonne représente le nombre de blocs de code. La colonne suivante représente le nombre de données de tests générés pour les 10 itérations de l'expérimentation. La cinquième colonne, elle-même séparée en trois, contient les résultats de l'expérimentation pour chacun des logiciels de génération utilisés (Testgen, Random et InKa). Les résultats listés dans ce groupe de trois colonnes représentent le pourcentage de la couverture de code de chaque code source testé. Dans leur article, les auteurs mentionnent que les résultats du logiciel Testgen proviennent de la littérature. D'autre part, le logiciel de tests Random est une implémentation d'un algorithme aléatoire de génération des données. Quant au logiciel Inka, il contient l'algorithme développé par les auteurs. Les résultats du tableau montrent qu'InKa est, à tout le moins, égal ou supérieur à Testgen et clairement supérieur à Random et qu'il est le seul générateur de données à couvrir entièrement tous les blocs de codes.

De leur côté, McGraw, Michael et Schatz [51] ont développé une méthode appelée « Génération dynamique des données de tests » qui a été implantée dans un logiciel appelé « Gadget ». Tout d'abord, le code source est instrumenté pour collecter des informations. Puis, le logiciel est exécuté plusieurs fois avec différentes données de départ. Les informations sont alors utilisées de façon heuristique pour déterminer si les tests générés satisfont un critère de couverture de branchement listé dans les spécifications du logiciel. En résumé, la méthode de génération automatique des données est basée sur le type des paramètres des fonctions et sur les conditions de branchements à l'intérieur de ces mêmes fonctions. L'utilisation de cette méthode accélère le processus pour déterminer les conditions aux frontières (section 2.1 : Conditions aux frontières). Cependant, elle

ne peut valider que le logiciel soit correct. L'humain doit déterminer si les valeurs obtenues par l'utilisation des valeurs générées automatiquement sont valides.

2.7 Patrons de tests

Le mot « pattern », traduit en français par « patron », exprime l'ensemble des éléments de solution utilisés par des spécialistes pour résoudre un type de problème dans un contexte donné [52]. Dans ce sens, la locution « patrons de tests » fait référence aux paires problèmes/solutions rencontrées dans la construction de tests de logiciel.

Clifton [53] définit 27 patrons de tests, regroupés sous 10 familles :

1. Succès/Échec: les plus simples des patrons de tests, ils indiquent si le résultat est celui qui était attendu;
2. Géré par les données : ces patrons séparent les données de tests du test lui-même. Il est alors plus aisé de modifier les données sans modifier le code informatique. Son utilisation principale est lors du développement de base de données. Cette méthode est explicitée plus avant dans la section 2.8 : Logiciel de tests;
3. Transaction de données : concerne tout ce qui est relié à la transmission des données et à leur persistance (exemple : base de données);
4. Gérer les collections : traite les collections (`list`, `vector`, `array`, `map`, etc.);
5. Performance : tous les patrons concernant la vitesse, la mémoire utilisée et tout ce qui est relié au domaine de la performance du logiciel;
6. Processus : groupe les méthodes pour tester les processus. En essence, un processus est une forme « d'unité logique »;
7. Simulation : les patrons reliés à la simulation de service pour permettre les tests de logiciel (exemple : la simulation d'une imprimante);
8. Fils d'exécution multiples : une des formes de tests les plus difficiles à effectuer en raison de la nature indépendante des fils d'exécution (*Multithreading*);
9. Test aux marges : groupe les méthodes reliées à la génération des conditions à la limite de marges d'opération du logiciel. Par exemple : manque de mémoire, saturation des canaux de communication, etc. (*Stress test*);
10. Couche de présentation : Les patrons de tests reliés à l'interface graphique.

Dans un langage de programmation objet, il existe principalement deux méthodes pour effectuer le patron de simulation[54;55] : l'objet bouchon (*mock object*) et l'objet autoterminé. L'objet bouchon simule le fonctionnement d'un matériel avec lequel le logiciel doit communiquer. Par exemple : un objet simulera une liaison avec une base de données en attendant que la base de données soit opérationnelle.

Dans la méthode de l'objet autoterminé, c'est cet objet même qui termine la chaîne de fonction. C'est-à-dire qu'il joue lui-même le rôle de « bouchon ». Ceci est simple et rapide mais mal adapté dans quelques cas :

1. Lorsque la classe testée communique avec plusieurs instances de la classe dont elle dépend;
2. Lorsque la classe testée peut détruire l'objet dont elle dépend;
3. Lorsqu'une autre classe de test nécessite l'écriture du même bouchon.

2.8 Logiciel de tests

Un logiciel de tests est un logiciel qui lit les données à tester dans un fichier d'entrée, les traite et écrit les résultats dans un fichier de sortie. Le fichier de données d'entrée peut être de type texte ou binaire. Il en est de même du fichier de sortie. De plus, le format de ces fichiers est déterminé par l'implémentation du logiciel de tests.

Un logiciel de tests peut être fabriqué à partir d'unité(s) de logiciel, de module(s) ou de bibliothèques(s). Aussi, il peut passer les valeurs lues à un module exécutable externe. Les valeurs lues seront alors des paramètres de cet exécutable. Le cas échéant, il peut recueillir les valeurs retournées de l'exécutable appelé et les écrire dans le fichier de sortie.

La comparaison des valeurs du fichier de sortie avec les résultats attendus confirme ou infirme les tests effectués. Cette validation du fichier de sortie permet de construire une hiérarchie de tests où chaque fichier de sortie doit être conforme

au résultat attendu pour que puisse être appelé le prochain test. De plus, puisque les données de tests sont dans des fichiers à l'extérieur du logiciel de tests, il est possible de modifier ces fichiers sans modifier le logiciel de tests.

2.9 Gestion des erreurs

Une gestion des erreurs efficace et précise est difficile à implémenter lors de la création d'un logiciel. Les messages d'erreurs doivent être générés aussi près que possible de la source d'erreur et être pertinents. Il existe principalement deux types d'erreurs : avec récupération et sans récupération. Les erreurs avec récupération sont celles qui permettent au logiciel de continuer son exécution. Par exemple, pour un analyseur lexical/sémantique, des erreurs lexicales, syntaxiques et sémantiques sont écrites dans les fichiers de tests. Ainsi, lors des tests, le type d'erreur et sa localisation seront répertoriés et le logiciel essayera de continuer. Un sémaphore global d'erreur sera activé. Le fichier de sortie de ces erreurs sera alors un fichier étalon pour la gestion des erreurs.

Les méthodes sans récupération sont probablement les plus simples à implémenter. Il suffit de faire remonter les fonctions fautives jusqu'à un endroit préalablement choisi. Dans un langage avec un mécanisme de traitement des exceptions, l'utilisation des fonctionnalités `throw` et `try/catch` permet de sortir des zones d'erreurs. L'exemple suivant montre les principes de base des fonctionnalités `throw` et `try/catch`.

Exemple 5 : Chemin d'appel des fonctions

```
CFOO:: ajouterClient(CClient oClient, CPath oPath)
{
   CString nom= ''CFOO::AjouterClient''; // Le nom de la
                                         // fonction
   oPath.ajouter(nom, oClient);          // Ajoute dans la
                                         // liste des
                                         // chemins les
                                         // informations
                                         // relatives au
client.
   try
   {
      // Ici le code qui génère une erreur

      // Si pas d'erreur
      // La fonction s'est terminée normalement,
      // Il faut donc enlever la méthode
      oPath.enlever(nom);
   }
   catch
   {
      throw oPath;
   }
}
```

Au début de la méthode, le nom de celle-ci est ajouté à l'objet `oPath` en utilisant l'objet `oClient`. Il est enlevé si la méthode se termine normalement. Dans le cas où il est inadéquat de copier l'objet contenant les données, les méthodes suivantes devraient être disponibles au programmeur : `oPath.ajouter(nom, NULL)` et `oPath.ajouter(nom)`.

Lorsque `CFOO:: ajouterClient(…)` lance l'objet `oPath`, il est de la responsabilité de la méthode qui reçoit cet objet de le traiter.

2.10 Revue de code

Une façon simple de vérifier que le code implémenté possède un nombre minimal d'erreurs est de faire une revue de code. Le code source est distribué aux

programmeurs. Ceux-ci le lisent et y trouvent des erreurs. Cependant, avec les logiciels qui atteignent maintenant les centaines de milliers de lignes de code ou même le million, cette méthode manuelle n'est plus efficace.

L'inspection automatique de logiciel [56] utilise un logiciel pour analyser le code. Des critères de décision tels que la complexité, le nombre de lignes de code, le nombre d'appels de fonctions et bien d'autres permettent de cibler des zones où la probabilité de trouver des bogues est plus grande. L'équipe de développement peut alors se concentrer sur ces zones pour optimiser les chances de découvrir des bogues.

2.11 Tests d'interfaces graphiques

Les interfaces graphiques avec leur grand nombre de boutons, de listes, de boîtes de texte et de boîtes de messages sont de réels défis à tester. Krasner et Pope [57] ont devisé le modèle : Modèle-Vue-Controlleur[16]. Ce modèle est devenu par la suite un paradigme de la programmation orientée objet.

Cependant, nombre de programmeurs font fi de ce paradigme, avec comme conséquence la construction de logiciels plus difficiles ou plus onéreux à tester.

Les méthodes et outils pour tester les interfaces graphiques forment à eux seuls un domaine de test. Linz et Daigl [58] ont résumé les propriétés que doivent avoir les outils logiciels pour bien tester ces interfaces :

1. Capture : capture les clics de la souris et les entrées faites au clavier dans les divers éléments de l'interface;
2. Programmable : les tests capturés sont écrits dans un langage qui est souvent ressemblant au langage Visual Basic;

[16] Modèle-Vue-Controlleur : architecture qui sépare le logiciel en trois entités. Modèle: le code source. Vue : affiche/transmet des données avec *Model*. *Controller* : donne les instructions à *View* pour l'affichage et à *Model* pour les calculs ou les entrées/sorties de données avec des fichiers.

3. Point d'arrêt : dans le but de déterminer si le logiciel sous test fonctionne correctement, le testeur peut insérer des points de contrôle dans les scripts capturés. Les valeurs des attributs des éléments de l'interface tels que : la couleur RBG, la position ou la grandeur sont alors comparées avec les valeurs attendues;
4. Ré-exécution : Une fois capturés, les scripts peuvent être relancés pour répéter les tests effectués, effectuant ainsi des tests de régression.

À partir de ces propriétés, les auteurs décrivent une méthode à employer pour tester les interfaces graphiques en utilisant des logiciels spécialisés de tests d'interface.

Beck [59] relate que, selon son expérience, les tests d'interfaces gérés par des scripts sont trop fragiles et ont tendance à souvent rapporter des erreurs d'interfaces. En réalité, il s'agit souvent de modifications cosmétiques de l'interface.

2.12 Logiciels types

Il existe de nombreux logiciels de tests, et chacun vise un langage particulier (Java, .Net, Perl, etc.). Le site de Cunningham [60] liste un bon nombre de logiciels de tests classés par thème. Plusieurs de ceux-ci sont disponibles sous licence *GNU General Public License* (*GNU GPL*) [61].

Une nouvelle tendance ces dernières années s'est dessinée avec l'introduction des logiciels de tests d'unités logiciel tels que ceux de la famille xUnit. Le tableau 11 liste quelques-unes de ces applications [37] et leur domaine d'utilisation :

Tableau 11 : Logiciels de tests

Logiciels	Domaines d'application
jUnit	Langage Java
jfcUnit	Interface graphique Java utilisant la bibliothèque *Swing*
nUnit	Environnement de développement .NET
nUnitForms	Interface graphique développée dans l'environnement .NET
oUnit	Base de données : Oracle
dbUnit	Base de données
sqlUnit	Environnement de développement pour exécuter des tests de régression sur des procédures stockées dans les bases de données
httpUnit	Applications Internet

Un extrait de l'exemple de code [62] qui montre le mode de fonctionnement interne de l'utilisation de jUnit, premier-né de la famille des xUnit, est présenté à l'exemple 6.

Exemple 6 : Implémentation avec jUnit

```
 1  Public class MoneyTest extends TestCase {
 2     //...
 3     public void testSimpleAdd()
 4     {
 5        Money m12CHF   = new Money(12, "CHF");       //
    (1)
 6        Money m14CHF   = new Money(14, "CHF");
 7        Money expected = new Money(26, "CHF");
 8        Money result   = m12CHF.add(m14CHF);         //
    (2)
 9
10        Assert.assertTrue(expected.equals(result));  //
    (3)
12      }
13  }
```

Pour définir une classe qui fera l'objet de tests, ici la classe `MoneyTest`, celle-ci doit être dérivée de la classe `TestCase` (ligne n°1). Tel que mentionné dans la documentation de jUnit, une méthode de test n'a pas de paramètre. Dans cet exemple, la méthode `testSimpleAdd(...)` exécutera le test de l'addition de deux montants d'argent (ligne n°3). Ces montants d'argent sont des objets (`m12CHF`[17] et `m14CHF`) de la classe `Money` (ligne n°5 et n°6). Le constructeur de la classe `Money` utilise deux paramètres : le montant et le type d'unité monétaire. Le type d'unité monétaire est spécifié sous forme de chaîne de caractères. La connaissance complète du fonctionnement des classes `Money` et `TestCase` n'est pas nécessaire pour le test qui suit.

La ligne n°7 montre la création d'un objet `expected`, de type `Money` qui sera utilisé comme étalon. La valeur de l'étalon est calculée par le programmeur et placée dans l'objet.

---

[17] CHF : abbréviation du latin *Confoederatio Helvetica* (confédération helvétique) et de Franc, autrement dit, des francs suisses. Le latin est utilisé pour préserver la neutralité parmi les communautés linguistiques de Suisse.

La ligne n°8 montre que l'objet `m12CHF`, de type `Money`, possède une méthode (`m12CHF.add(...)`) qui reçoit comme paramètre un objet de type `Money` et qui retourne un objet de type `Money`. Ceci permet de présumer que la méthode ajoute ce montant à la valeur interne et retourne la valeur calculée, et ce, sans changer la valeur interne.

À la ligne n°10, il y a trois objets : `Assert`, `expected` et `result`. L'objet `Assert` fait partie d'une classe fournie dans jUnit et est déclarée statique. Il est donc accessible même s'il est non déclaré dans la méthode `testSimpleAdd()`. La méthode `Assert.assertTrue(...)` est une méthode qui permet de valider que le paramètre passé soit vrai. L'objet `expected` est de classe `Money` et possède une méthode `Money.equals(Money)` qui reçoit l'objet `result`, de type `Money`, et qui retourne une valeur booléenne. La valeur sera vraie si l'objet `expected`, qui reçoit le paramètre, et le paramètre lui-même, l'objet `result`, sont égaux. La fonction retournera faux autrement.

Dans cet exemple, il est important de souligner les points suivants :

1. Création d'une classe MoneyTest pour tester la classe Money;
2. Création de quatre objets de type Money (m12CHF, m14CHF, expected et result);
3. Utilisation de Assert.assertTrue(...) comme comparateur entre le résultat et l'étalon.

Cette forme d'implémentation est une façon simple d'effectuer des tests. Cependant, dans cet exemple, la méthode `testSimpleAdd( )` crée trois objets `Money` qui seront nettoyés par le *garbage collector* de Java. De même, l'utilisation des fonctions de test de jUnit ne permet pas d'accepter directement des paramètres. Néanmoins, il existe des fonctionnalités dans jUnit qui permettent de contourner cette limitation. Le lecteur désireux d'en connaître plus se rapportera à la documentation de jUnit.

3

MÉTHODE GÉNÉRIQUE DE GESTION DES TESTS D'UNITÉS LOGICIEL

La méthode générique proposée pour la gestion des tests d'unités logiciel est basée sur la méthode de test : « Logiciel de tests » (section 2.8). La méthode proposée permet de gérer les cas d'utilisation séquentielle ou simultanée de plusieurs logiciels de tests. De même, elle s'applique aux cas où les fichiers de sortie sont texte ou binaire. Chaque exécution de tests génère un rapport de test et celui-ci est traité selon le succès ou l'échec du test en question.

L'architecture de la méthode de gestion générique de tests permet d'intégrer les différentes unités logiciel en un tout cohérent. Elle permet de découpler le logiciel de tests, les fichiers de tests et les composants logiciel. Cette architecture permet d'appeler toutes les fonctions accessibles avec un seul niveau d'interface, à l'exception des méthodes privées de la programmation orientée objet. Ces dernières ne pouvant être accédé par l'extérieur de l'objet. Dans ce cas, il est nécessaire de valider que les données suivent le flot de contrôle depuis le point d'entrée (une méthode publique) jusqu'à la méthode désirée (méthode privée).

Les sections de ce chapitre contiennent les éléments nécessaires pour la compréhension et l'utilisation de la méthode proposée. Tout d'abord, la description de la méthode proposée sera présentée. Suivra la section « Preuve de concept » qui décrit les différentes implémentations qui ont amené à l'élaboration de la méthode proposée. Cette section est présentée ici car les principes sous-jacents dans ces implémentations peuvent être utiles pour la compréhension. De même, cette section contient le schéma de la base de données qui a été élaborée à partir

de cette preuve de concept. Suivra une section décrivant un prototype opérationnel de la méthode proposée. Pour terminer, des remarques sur une expérimentation in situ, une discussion et une liste de développements futurs seront présentées.

3.1 Description de la méthode de gestion

La notion de logiciel de tests a été introduite à la section 2.8 : Logiciel de tests. Il s'agit d'un logiciel construit autour de composants devant être testés. Ce logiciel a donc pour but de tester les fonctionnalités de chaque composant. Il est avantageux de centraliser les tests d'un composant dans un logiciel de tests. Le facteur principal qui influence la décision de construire un ou plusieurs logiciels de tests pour un composant est l'importance « critique » des fonctions de celui-ci. Plus les fonctions implémentées sont critiques, plus la clarté et la simplicité de l'implémentation des tests doivent être présentes. Donc, il s'agit de diminuer la complexité du logiciel de tests. Si celle-ci est trop grande, le coût de maintenance du logiciel de tests sera d'autant plus grand.

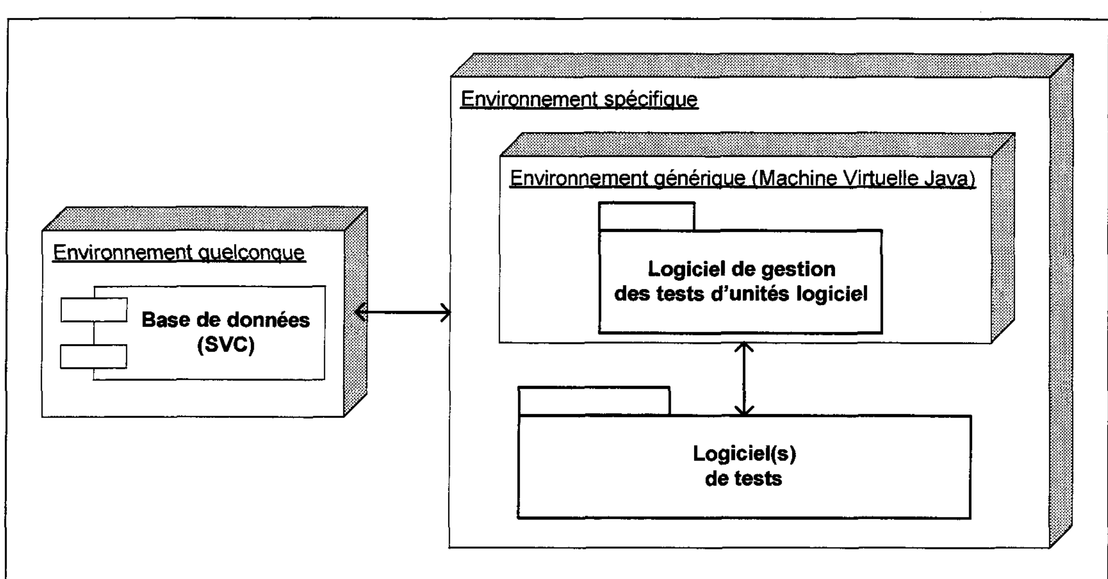
D'autre part, un des intérêts de l'approche par logiciel de tests est qu'une fois construit, il peut être ré-exécuté afin de s'assurer que l'aspect testé du composant reste valide. Ainsi, chaque fois qu'une modification est apportée à un composant, le logiciel de tests ré-exécuté s'assure que les autres fonctionnalités déjà en place restent opérationnelles. C'est la notion même de tests de régression, introduite à la section 1.2.5 : Test de régression. Pour être efficace, l'approche par logiciel de tests demande une gestion précise des divers éléments la constituant.

3.1.1 Environnement

La généricité de la méthode proposée s'appuie sur une utilisation multiplateforme du logiciel de gestion des tests et sur une méthode de programmation applicable dans tous les environnements où la lecture/écriture de

fichiers est permise. L'exécution multiplateforme du logiciel de gestion est atteinte grâce à son implémentation en langage Java. La figure 17 illustre la cohésion entre les environnements dans lesquels le logiciel de gestion aura des interactions.

Figure 17 : Cohésion entre les environnements



Cette méthode est conçue pour travailler dans un contexte avec trois environnements. Le premier environnement est celui de la base de données SVC (Système de Versions Concurrentes). Il est à noter que la base de données SVC est dans un environnement quelconque. Il faut entendre par « environnement quelconque » un environnement qui permet d'utiliser une base de données et un lien de communication pour y accéder.

Le second environnement est celui du logiciel de tests. Cet environnement est spécifique[18] à chaque logiciel de tests. C'est-à-dire que, si un logiciel de tests est compilé pour un environnement spécifique, par exemple en C++ sous Linux, il ne sera exécutable que dans cet environnement. Le cas d'exception où le logiciel

[18] Il est à noter qu'un projet informatique peut consister à développer des logiciels dans différents environnements. Chacun de ces environnements est alors spécifique pour le logiciel en question.

de tests est implémenté en Java ne pose aucune difficulté, car l'environnement spécifique devient simplement l'environnement que le programmeur choisit pour faire le développement.

Le troisième environnement est celui du logiciel de gestion des tests. En fait, il s'agit d'un environnement générique grâce à l'utilisation d'une machine virtuelle Java donc, en pratique, cet environnement générique est compatible avec l'environnement spécifique du logiciel de tests.

3.1.2 Éléments de la méthode

Cette section contient les concepts nécessaires pour décrire le fonctionnement de la méthode de gestion des tests unitaire. Le terme « élément » est utilisé pour bien montrer qu'il s'agit de concepts d'information ou de regroupements d'informations reliées. À ce stade du développement de la méthode, il est adéquat de garder un sens conceptuel plutôt que d'utiliser des termes qui s'inspirent d'une implémentation spécifique.

3.1.2.2 Élément : project

Un élément *project* contient toutes les informations requises pour gérer celui-ci : numéro de référence, description, nom du responsable et adresse courriel de ce dernier. De plus, un élément *project* peut contenir toute combinaison d'éléments *software* et *groupSoftware*, pour autant qu'au moins un de ces éléments soit présent.

3.1.2.3 Élément : groupSoftware

L'utilisation d'un élément *groupSoftware* est requise lorsque deux ou plusieurs logiciels (éléments *software*) doivent être exécutés simultanément. Par exemple, pour tester le fonctionnement d'un protocole de communication, il est nécessaire que le client et le serveur soient activés. Cet élément contient les

informations permettant de le gérer : la description du groupe de logiciels, les éléments *software* requis et l'adresse courriel où envoyer le rapport de tests.

Aussi, un élément *groupSoftware* peut contenir zéro, un ou plusieurs éléments *preActionTest* ou *postActionTest*. Ces éléments sont, respectivement, des actions à exécuter avant et après l'exécution du *groupSoftware*.

3.1.2.4 Élément : software

L'élément *software* contient les informations relatives à l'exécution d'un logiciel de tests. Ces éléments sont : la description du logiciel de tests, le numéro de référence du logiciel (version), l'adresse courriel de destination du rapport d'exécution du test, la localisation de l'exécutable qui sera testé ou la localisation de l'archive (fichier .zip) le contenant et au moins un élément *test*. Dans le cas où le logiciel de tests génère un fichier générique de sortie*, la localisation de celui-ci et la localisation du fichier étalon correspondant sont présentes dans l'élément *software*.

Aussi, de l'information est conservée concernant le type de système d'exploitation pour lequel le logiciel à tester est construit. Cette information permet d'exécuter le logiciel sous tests seulement dans l'environnement pour lequel il a été conçu.

Si un élément *software* fait partie directement d'un élément *project*, il doit contenir au moins un élément *test*. S'il possède plusieurs éléments *test*, ceux-ci seront exécutés séquentiellement. Si l'élément *software* fait partie d'un élément *groupSoftware*, cet élément *software* ne peut avoir qu'un seul élément *test*. Dans le cas contraire, si deux élément *software* appartenant à un *groupSoftware* pouvait avoir deux tests ou plus, il serait difficile de synchroniser les éléments *tests* subséquents au premier élément *test*.

Pour faire plusieurs tests avec les mêmes logiciels contenus dans un élément *groupSoftware*, il est nécessaire de dupliquer l'élément *groupSoftware* et

ses éléments *software*. Toutefois, les éléments *test* seront modifiés pour traiter le nouveau test à exécuter.

3.1.2.5 Élément : test

Un élément *test* contient les informations qui décrivent le test à effectuer. Celles-ci sont : la description du test, la localisation du ou des fichiers de données d'entrée, la localisation du ou des fichiers de sortie ainsi que la localisation du ou des fichiers étalons correspondants. De plus, il est possible de spécifier les arguments qui seront fournis au logiciel de tests. Un élément *test* peut contenir zéro, un ou plusieurs éléments *preActionTest* ou *postActionTest*.

3.1.2.6 Élément : preActionTest

Un élément *preActionTest* contient les actions à prendre avant d'effectuer un test. Ces actions peuvent inclure la mise en place de variables d'environnement ou la mise en place de données dans une banque de données.

3.1.2.7 Élément : postActionTest

Un élément *postActionTest* contient les actions à prendre après que le test ait été effectué. Des exemples de ces actions sont : retirer les variables d'environnement déclarées précédemment par un élément *preActionTest* ou nettoyer une base de données après l'exécution d'un test.

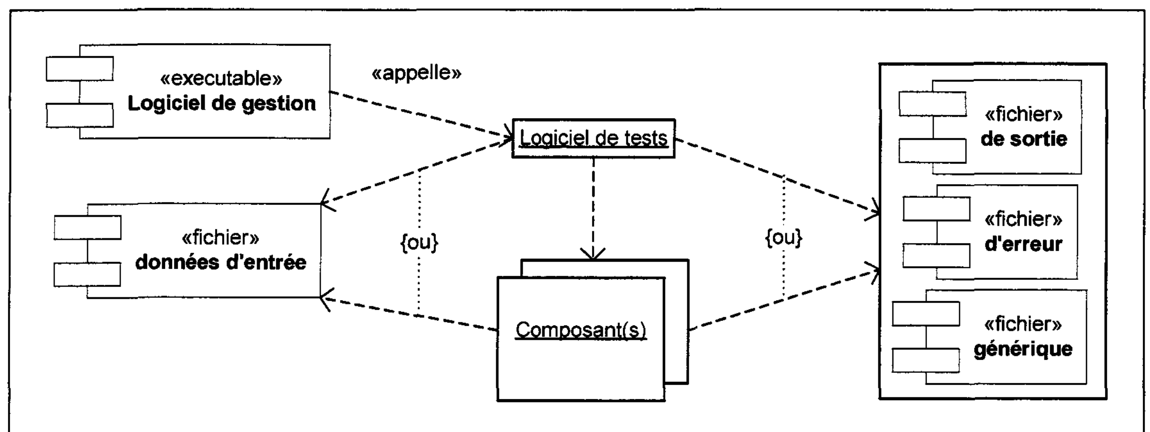
3.1.3 Résultats des tests

Les résultats des tests sont transmis, sous forme de rapport de tests (section 3.3.3 : Rapport de tests), aux personnes désignées dans les éléments project, *groupSoftware* et *software*. Le contenu de ces rapports est tributaire du résultat de l'exécution du test de régression.

3.1.4 Spécifications du logiciel de tests

Les spécifications du logiciel de tests décrivent comment doit être organisé le logiciel de tests pour s'intégrer dans la méthode de gestion proposée. La figure 18 montre l'architecture d'un typique logiciel de tests.

Figure 18 : Architecture d'un logiciel de tests



1. **Logiciel de gestion** : le logiciel de gestion appelle le logiciel de tests avec les paramètres qui permettront au logiciel de tests de lire les fichiers d'entrée et d'écrire les fichiers de sortie requis.
2. **Fichier de données d'entrée** : contient les tests qui seront exécutés.
3. **Logiciel de tests** : est exécuté avec les paramètres reçus. Il existe deux cas de figure. Le premier est celui où le nom du fichier de données à traiter est passé à une fonction du composant à tester. De même, le nom du fichier de sortie pourrait être passé à une autre fonction du composant qui, elle, s'occuperait d'écrire le fichier de sortie. Dans le second cas d'utilisation, les données lues dans le fichier d'entrée sont utilisées par le logiciel de tests comme paramètres pour les fonctions à tester. Une utilisation intéressante des données fournies au logiciel de tests est de créer des pré/post conditions, c'est-à-dire placer le logiciel de tests dans un état particulier avant le test ou de vider le contenu de certaines variables après le test. Par exemple : initialiser une base de données avec des valeurs prédéterminées avant de tester le composant d'exécution de requête, puis vider la base de données.
4. **Composant(s)** : contient l'ensemble des fonctions à tester. Il peut être statique, dynamique ou exécutable.
5. **Fichier de sortie** : contient les résultats du test.
6. **Fichier d'erreur** : contient les erreurs rencontrées en cours de tests. Le modèle de gestion des erreurs de l'unité et du logiciel de tests décidera si le

fichier d'erreur et le fichier de sortie sont un seul et même fichier. Il est cependant usuel de faire la gestion des erreurs de façon séparée.

7. **Fichier générique** : contient des données qui devraient être identiques pour tous les tests d'un même logiciel, dans le cas où un fichier générique est produit. L'exemple le plus courant est un fichier qui liste les fuites de mémoire.

Pour terminer, le code source d'un logiciel de tests devrait, à tout le moins, contenir en en-tête les informations suivantes :

1. Description du fichier de tests;
2. Description précise du format du fichier d'entrée;
3. Description précise du format du fichier de sortie;
4. Un exemple des données de tests.

Un exemple de logiciel de tests est présenté à l'annexe 2.

3.1.4.1 Composant statique/dynamique

Un composant statique est un fichier compilé, ou une bibliothèque, dont les éléments utilisés par le logiciel développé sont intégrés lors de compilation.

Un composant dynamique consiste en une bibliothèque de fonctions dont les éléments seront utilisés par un autre logiciel lors de l'exécution de celui-ci. Un des types les plus connus est le « .DLL » (*Dynamic Link Library*).

3.1.4.2 Composant Exécutable

Dans le cas où le composant à tester est un fichier exécutable, l'utilisation d'un logiciel de tests n'est pas nécessaire et il est alors possible de traiter ce logiciel comme un logiciel de tests. Toutefois, ceci est possible pourvu que le fichier exécutable respecte les conditions nécessaires pour être considéré comme un logiciel de tests par le logiciel de gestion, c'est à dire :

1. Accepter des paramètres en arguments;
2. Lire un fichier de données;
3. Écrire un fichier de données;
4. Pouvoir'autoterminer.

De plus, la méthode proposée permet de grouper deux logiciels ou plus avec leurs fichiers de tests correspondants. L'utilisation la plus directe de cette particularité concerne les tests reliés aux protocoles de communication : deux logiciels doivent travailler en simultané pour se transmettre des données.

3.1.4.3 Fichier de tests

Les fichiers de tests sont définis comme étant les fichiers d'entrée de données. Le logiciel de tests connaît le format du fichier de tests. Dans le but de simplifier la lecture des données de tests, il est suggéré qu'il n'y ait aucune autre information dans les fichiers que les données. S'il y a lieu, les commentaires sont dans le logiciel de tests correspondant. Le format du fichier de tests est décrit dans le logiciel de tests.

3.1.4.3.1 Paramétrage des fonctions appelées

Un logiciel de tests appelle des fonctions pour leur fournir des données et il s'attend à recevoir des résultats. Les données sont passées par les paramètres des fonctions. Le cas où la fonction ne possède pas de paramètre est un cas particulier des fonctions avec paramètres.

Pour appeler une fonction, il est nécessaire d'assigner une valeur à chaque paramètre utilisé. Les paramètres sont essentiellement de deux types : type simple ou type complexe. Un type simple est défini comme un type de base, c'est-à-dire les nombres entiers et les nombres réels, signés ou non, ainsi que les chaînes de caractères. Les types complexes sont des structures de données ou des objets.

3.1.4.3.2 Type simple

Usuellement, dans un fichier de tests, la donnée le plus à gauche est celle qui servira à diriger les appels de fonction par l'utilisation de la fonction *switch* ( ) ou d'une autre façon similaire. Souvent, les fonctions utilisent des paramètres, des

variables globales ou des variables d'environnement. Les valeurs de ces paramètres ou variables peuvent être placées dans le fichier de tests ou faire partie d'un état de l'environnement. Les possibilités de cette méthode sont sans limites, car la construction de ces fichiers de tests est chaque fois basée sur le besoin de la fonction à tester. Voici un exemple simple de fichier de tests pour tester des fonctions qui effectuent des opérations mathématiques de base :

Exemple 7 : Fichier d'entrée (type simple)

```
+  1  2
-  4  4
/  1  0
*  40 1000
+  0  0
```

Dont les champs sont :

- 1er champ : le signe de l'opérateur, utilisé pour sélectionner la fonction appelée;
- 2e champ : le premier paramètre de la fonction appelée;
- 3e champ : le deuxième paramètre de la fonction appelée.

Dans l'exemple ci-dessus, il est possible d'ajouter un quatrième champ, qui serait la valeur retournée par l'appel de la fonction. Par exemple :

Exemple 8 : Fichier de tests (type simple : bis)

```
+ 1 2 3
- 4 4 0
/ 1 0 I
* 40 1000 40000
+ 0 0 0
```

Dans ce cas, le travail de validation est effectué par le logiciel de tests en comparant la valeur retournée par la fonction testée avec la valeur attendue[19]. La

[19] Note : C'est à cet endroit que la méthode utilisée par les *xUnit* et celle du logiciel de tests se recoupent. Chacune utilise deux valeurs et la valeur de retour. Le grand avantage des xUnit est d'avoir des outils de développement matures et un grand bassin d'utilisateurs. En contrepartie, le logiciel de tests possède une grande facilité à ajouter/modifier des tests sans modifier le code

valeur `I` indique qu'il est impossible d'effectuer une division par zéro. Le traitement de ce type d'erreur est fonction du langage de programmation utilisé. Pour ajouter un test, il suffit d'ajouter une ligne de données au fichier.

3.1.4.3.3 Type complexe

Dans les cas où la fonction à tester requiert une donnée complexe, par exemple un objet ou une structure, ou une chaîne de caractères (assemblage d'une requête SQL), ces éléments peuvent être construits à partir d'éléments simples contenus dans le fichier de tests[20].

Exemple 9 : Fichier de tests (type complexe)

```
I
T Tremblay Jean 0
A Tremblay Jean 1
T Tremblay Jean 1
T Tremblay Jean 0
E Tremblay Jean 1
T Tremblay Jean 0
V
```

L'exemple plus haut contient les données pour tester une application travaillant avec une base de données. Les champs, placés sur une ligne, contiennent les valeurs suivantes :

(page précédente)
source. En effet, ceux-ci sont dans un fichier externe au logiciel compilé et chaque ajout ne demande pas une recompilation des codes sources.

[20] Il est aussi possible d'inscrire directement la requête SQL dans le fichier de tests. On pourrait dire alors qu'il s'agit d'un type simple.

Tableau 12 : Description de champ

Champ	Description
1	Le code de l'opération à effectuer
2	Le nom de famille
3	Le prénom
4	Le résultat de l'opération

La convention des codes d'opération est la suivante :

- `I` : initialisation de la base de données,
- `A` : ajoute un enregistrement,
- `T` : trouve un enregistrement,
- `E` : efface un enregistrement,
- `V` : vide la base de données.

Avec la convention des résultats :

- `1` : vrai
- `0` : faux

Selon le code d'opération, le logiciel de tests exécutera l'appel à la fonction correspondante et vérifiera la réponse attendue (celle du fichier) avec la réponse fournie par la base de données. Dans le cas où le composant à tester utilise des objets comme paramètres, il est possible au logiciel de tests de construire ceux-ci avec des données du fichier de tests puis de passer ces objets en paramètres.

3.1.5 Structure de répertoires

L'utilisation d'une structure de répertoires permettant un classement des informations des tests est recommandée. Le tableau 13 présente une structure de répertoire basée sur celle suggérée par Morin [63].

Tableau 13 : Structure de répertoires suggérée

Répertoire	Description
.\monProjet\	Base du projet
.\monProjet\bin	Produit(s) compilé(s)
.\monProjet\doc	Documentation du produit
.\monProjet\inc	Fichiers d'en-têtes
.\monProjet\lib	Bibliothèques externes
.\monProjet\prj	Fichiers de gestion du projet (*workspace*, *makefile*, etc.)
.\monProjet\src	Sources du projet
.\monProjet\test	Base des répertoires[21] de tests
.\monProjet\test\entree	Fichiers d'entrée de données
.\monProjet\test\etalon	Fichiers étalons (pour comparaison)
.\monProjet\test\rapport	Rapport des tests de régression
.\monProjet\test\sortie	Fichiers de sortie de données
.\monProjet\tmp	Fichiers temporaires de la compilation
.\monProjet\use	Importation des produits externes

Le répertoire « *use* » est utilisé pour contenir les produits d'autres projets. Par exemple, supposons qu'une compagnie a développé une bibliothèque de calcul et que cette bibliothèque soit un standard. Il importe donc de localiser adéquatement cette bibliothèque dans les environnements de développement de chaque projet informatique. De même, advenant une mise à jour de cette bibliothèque, les tests qui seront refaits devront l'être avec cette nouvelle bibliothèque.

[21] Il est possible de fonctionner sans sous-répertoire d'entrée, sortie, étalon et rapport. Dans ce cas, les noms de ces fichiers devront différer.

En localisant ainsi les utilisations de bibliothèques externes, le programmeur est assuré d'avoir toujours la version courante de la bibliothèque. Ceci suppose que le SVC est capable de gérer cette dépendance.

Il est hors de la portée de ce document de suggérer des méthodes de gestion de la configuration ou de numérotation de projets. La gestion de la configuration est habituellement déléguée à un SVC alors que la numérotation des projets est propre à chaque entreprise et peut être aussi simple ou complexe que désirée.

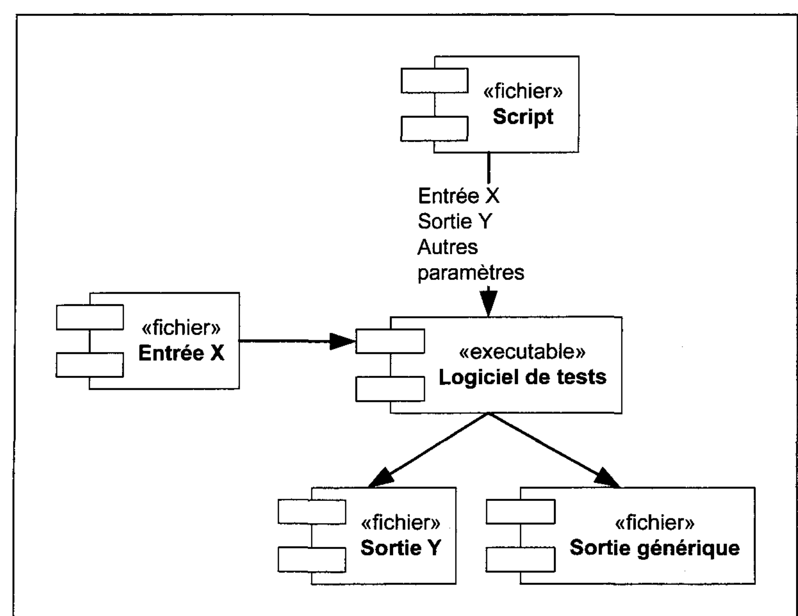
3.2 Preuve de concept

Des prototypes ont été réalisés pour valider le concept. Le but de ceux-ci était de valider la structure d'information qui soutient la méthode de gestion proposée.

3.2.1 Implémentation : mode console

L'implémentation initiale a été effectuée en mode console, en utilisant des fichiers script (DOS) : voir l'annexe 3 : FICHIER SCRIPT. La figure 19 montre les relations entre les composants qui sont utilisés dans le script.

Figure 19 : Écriture de fichiers de sortie (Script)



Le script appelle le logiciel de tests avec les paramètres suivants :

- Entrée X : localisation du fichier d'entrée;
- Sortie Y : localisation du fichier de sortie;
- Autres paramètres : tous les paramètres pouvant être utiles au logiciel :
  - i. Sémaphore pour l'écriture du fichier de générique de sortie;
  - ii. Options utilisées pour gérer l'environnement interne du logiciel de tests;
  - iii. Etc.

La liste « autres paramètres » peut être vide. Voici un exemple, tiré de la même annexe, dans lequel la symétrie des appels simplifie la gestion des fichiers.

Exemple 10 : Fichier script

```
rodin3 scheme0 16>..\out\t0_16.txt>nul
rodin3 scheme1 16>..\out\t1_16.txt>nul
```

Avec :

- Logiciel de tests : rodin3
- Fichier d'entrée : scheme0
- Autre paramètre : 16
- Fichier de sortie : « >..\out\t0_16.txt[22] »

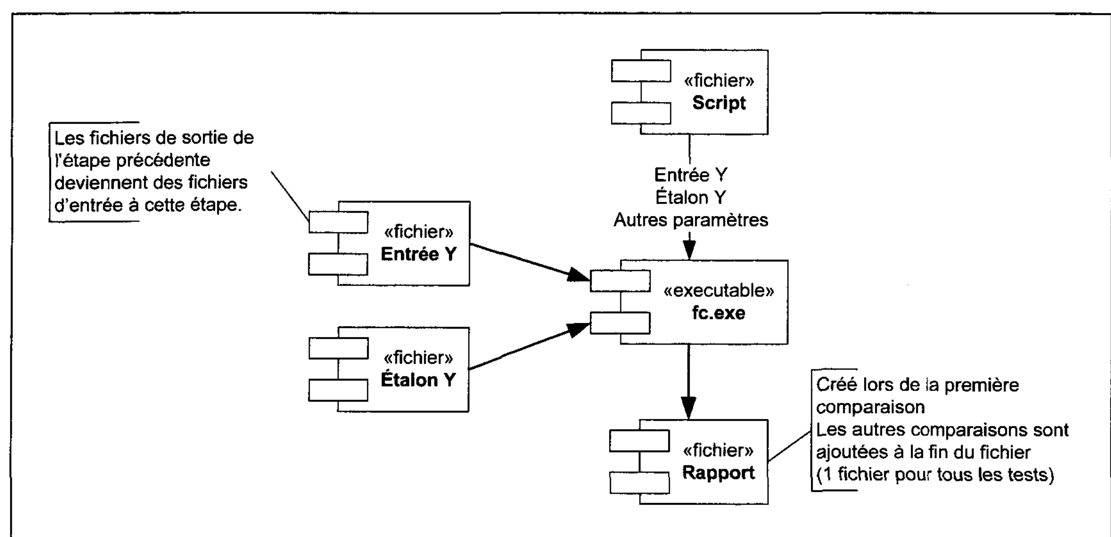
Le nom de fichier « t0_16.txt » est un acronyme décrivant le contenu : test du fichier scheme0 pour le paramètre 16 et ainsi de suite. Dans cet exemple, il est sans importance de connaître la signification du nombre 16.

Une ligne de script se termine par une redirection de l'écran vers « nul », donc il n'y a pas d'affichage à l'écran. Le même processus est appliqué pour tous les fichiers tests. Pour simplifier la discussion, le raisonnement avec le fichier générique de sortie sera omis. Toutefois, son fonctionnement est similaire à celui présenté ici.

La deuxième partie du script utilise les fichiers générés comme fichiers d'entrée.

[22] Les adeptes du mode commande reconnaîtront la redirection de la sortie, effectuée sur la sortie standard (l'écran par défaut) par le logiciel de tests, vers un fichier.

Figure 20 : Comparaison fichiers de sortie versus fichiers étalons



Voici un extrait de ce script :

Exemple 11 : Script (comparaison de fichier)

```
fc ..\out\schem* ..\out\etalon\* >>..\out\etalon\batchres.txt
```

Avec :

- Logiciel de comparaison : `fc`
- Fichiers d'entrée : `..\out\schem*`
- Fichiers étalons : `..\out\etalon*`
- Rapport de comparaison : « `>>..\out\etalon\batchres.txt` »

À remarquer l'utilisation de `fc`, un utilitaire de comparaison de fichiers qui fait partie du système d'opération, et de `*` un caractère générique (*wildcard*) qui permet de simplifier l'écriture de toutes les comparaisons. L'utilisation du caractère générique permet de comparer tous les fichiers débutant par `schem`, dans le répertoire `\out`, avec les fichiers correspondants, dans le répertoire `\etalon\`. Voici un extrait du contenu du fichier rapport généré dans un système d'opération en anglais :

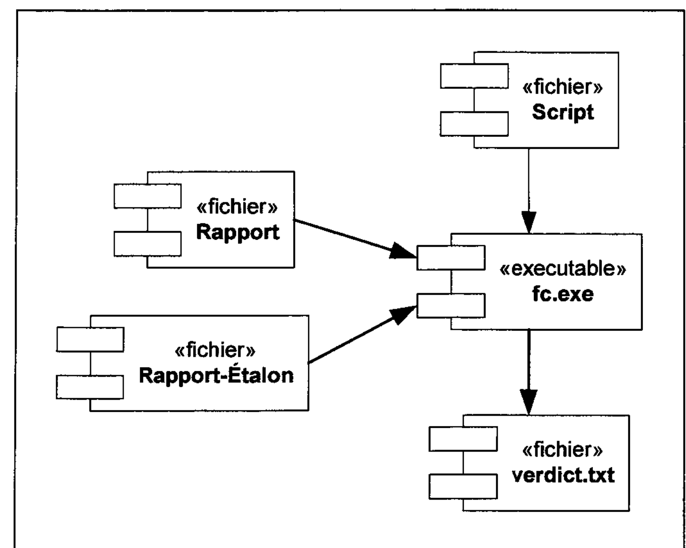
Exemple 12 : Rapport de tests (partiel)

```
Comparing files
r:\rodin\test\out\scheme0 and r:\rodin\test\out\etalon\scheme0
FC: no differences encountered
```

Donc, la comparaison entre le fichier de sortie et le fichier étalon correspondant ne génère aucune différence. Idéalement, tous les tests ont produit le même résultat, c'est-à-dire qu'il n'y a pas de différence rencontrée et les résultats des comparaisons sont inscrits dans ce fichier rapport.

La dernière partie du fichier de script consiste à comparer le fichier rapport avec le fichier rapport-étalon. La figure 21 montre le processus des flux d'information.

Figure 21 : Comparaison fichier rapport versus fichier rapport-étalon



Le fonctionnement de cette partie est similaire à celui de la deuxième partie. Cependant, le résultat final de tous les tests est placé dans le fichier « verdict.txt » dont voici le contenu (le rapport-étalon se nomme « maitre.txt ») :

Exemple 13 : Fichier verdict.txt

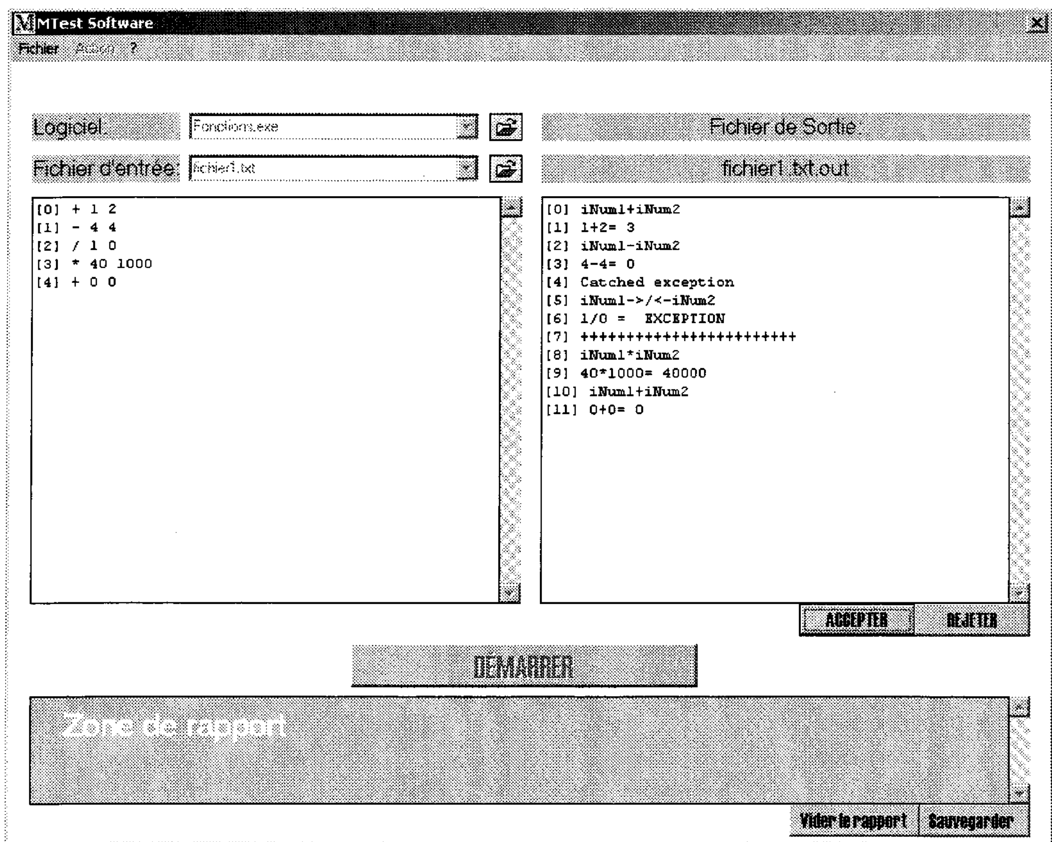
```
Comparaison des rapport.txt et maitre.txt
FC : aucune différence trouvée
```

Si dans la séquence d'action il y avait eu une différence entre un fichier de sortie et un fichier étalon, cette différence serait apparue dans le fichier « verdict.txt ».

3.2.2 Implémentation : mode graphique

À la suite de la preuve de concept en mode console, une preuve de concept en mode graphique a été construite. Celle-ci a permis d'établir le mode de fonctionnement qui simplifierait le travail du programmeur ainsi que de la structure d'information sous-jacente. Conformément à la méthode de développement par prototype, le prototype de la preuve de concept en mode graphique a été développé dans un langage simple, rapide et destiné à être jeté.

Figure 22 : Interface du prototype de la preuve de concept



La figure 22 montre l'état de l'interface après l'exécution du test du logiciel. Voici les actions qui ont été effectuées pour parvenir à l'état montré ci-dessus.

1. Sélection du logiciel : fonctions.exe[23];
2. Sélection du fichier de tests : fichier1.txt;
3. Activation du bouton DÉMARRER.

Les deux zones principales représentent :

- À gauche : le contenu du fichier d'entrée;
- À droite : le contenu du fichier de sortie.

---

[23] Note : Dans le cas où il y aurait des paramètres ou des arguments, ceux-ci seraient placés après le nom de l'exécutable.

Les boutons `ACCEPTER` et `REJETER` se trouvent sous le fichier de sortie. Le bouton `ACCEPTER` permet de muter le fichier affiché en sortie en fichier étalon. Le bouton `REJETER` rejette le fichier et permet de passer à une autre vérification de fichier de tests.

La zone de rapport sera utilisée dans une action ultérieure pour un fonctionnement en mode de test de régression, c'est-à-dire que le logiciel comparera le fichier de sortie et le fichier étalon et affichera dans cette zone le résultat de la comparaison.

L'option `Action`, de la barre de menu, permet de choisir parmi les actions suivantes :

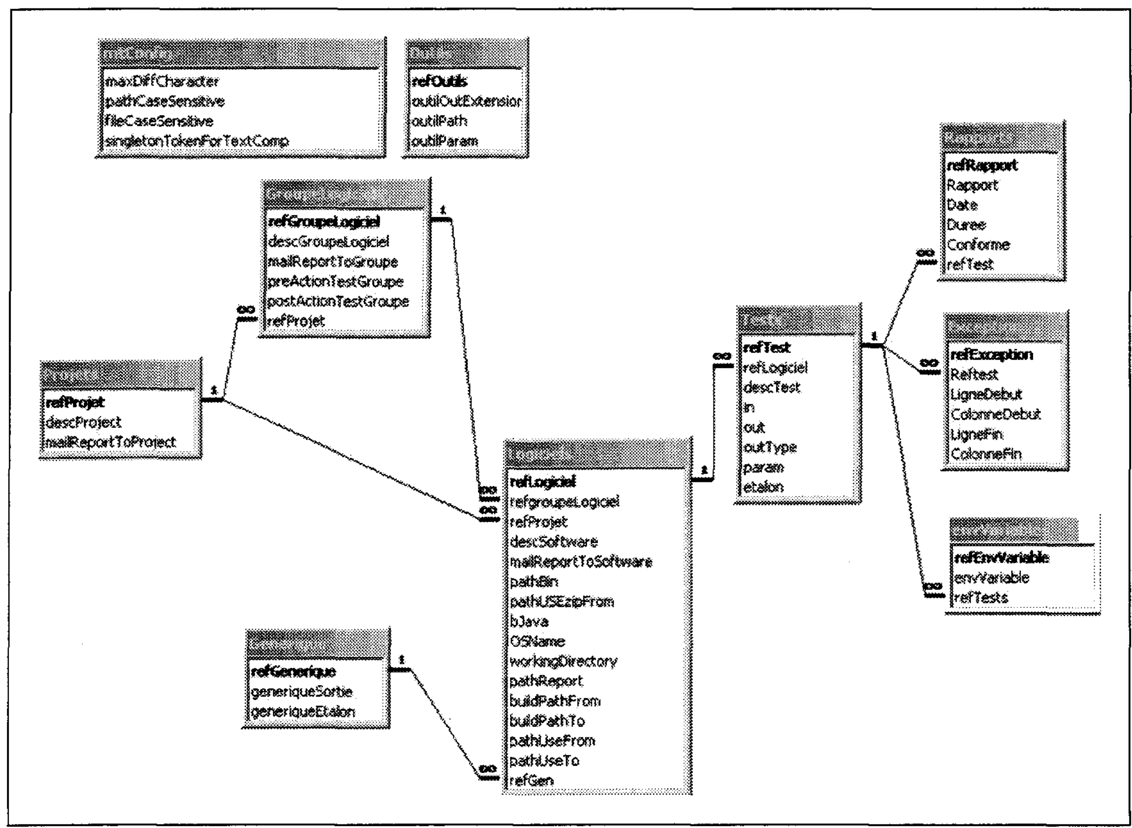
1. Tester un logiciel avec un fichier d'entrée;
2. Tester un logiciel avec tous ses fichiers d'entrée;
3. Tester tous les logiciels avec tous leurs fichiers d'entrée.

L'option `Action` est accessible lorsqu'un logiciel et un fichier d'entrée ont été sélectionnés et apparaissent dans les boîtes de textes appropriées.

3.2.3 Base de données

Pour faciliter la simulation de l'utilisation d'une base de données, un fichier XML est utilisé comme prototype de base de données. Le fichier utilisé est présenté à l'annexe 4 : Fichier de données XML. Ce fichier est une version simplifiée du schéma présenté à la figure 23.

Figure 23 : Schéma entité relation des données



Ce schéma montre les relations entre les différentes tables dont voici les définitions :

- mtConfig : diverses options de configuration;
- outils : informations reliées à l'utilisation automatique d'outils à appliquer sur le fichier de sortie;
- Projets : gestion des projets;
- GroupeLogiciels : liste les logiciels qui doivent être testés simultanément;
- Logiciels : informations reliées au logiciel sous test;
- Generique : gestion des informations des fichiers génériques;
- Tests : informations reliées à un test;
- Rapports : informations concernant les résultats des tests;
- Exception : localise, dans un fichier texte, les exceptions à la comparaison identique;
- envVariable : liste les variables d'environnement qui doivent être définies avant l'exécution d'un test.

Le lecteur remarquera que l'implantation XML dans le prototype de la méthode proposée et ce schéma sont différents. Cette différence, en plus d'être linguistique, est le résultat d'une implémentation sous forme de prototype. Le but important étant de faire un prototype pouvant permettre une implémentation incrémentale des fonctionnalités. Les noms dans le fichier XML sont autodescriptifs de même que ceux dans le schéma d'analyse de la base de données.

3.3 Prototype de la méthode proposée

À la suite de la preuve de concept en mode console puis à celle en mode graphique, la méthode proposée a été implémentée dans un prototype. Celui-ci a permis de valider les informations requises pour gérer les tests et de déterminer les informations requises par une interface graphique. La prochaine étape est de construire, en mode graphique, un logiciel opérationnel avec une conception la plus épurée possible. Il est important de garder en tête que la méthode doit être applicable aux petites entreprises et que la charge d'information ou de travail pour gérer les tests doit être minimale.

Dans ce sens, le prototype est dans la lignée de cette citation attribuée à Antoine de Saint-Exupéry [64] : *la perfection est atteinte, non pas lorsqu'il n'y a plus rien à ajouter, mais lorsqu'il n'y a plus rien à retirer.*

Conséquemment, le prototype effectuera les opérations de base sans chercher à pousser les options jusqu'à leur limite.

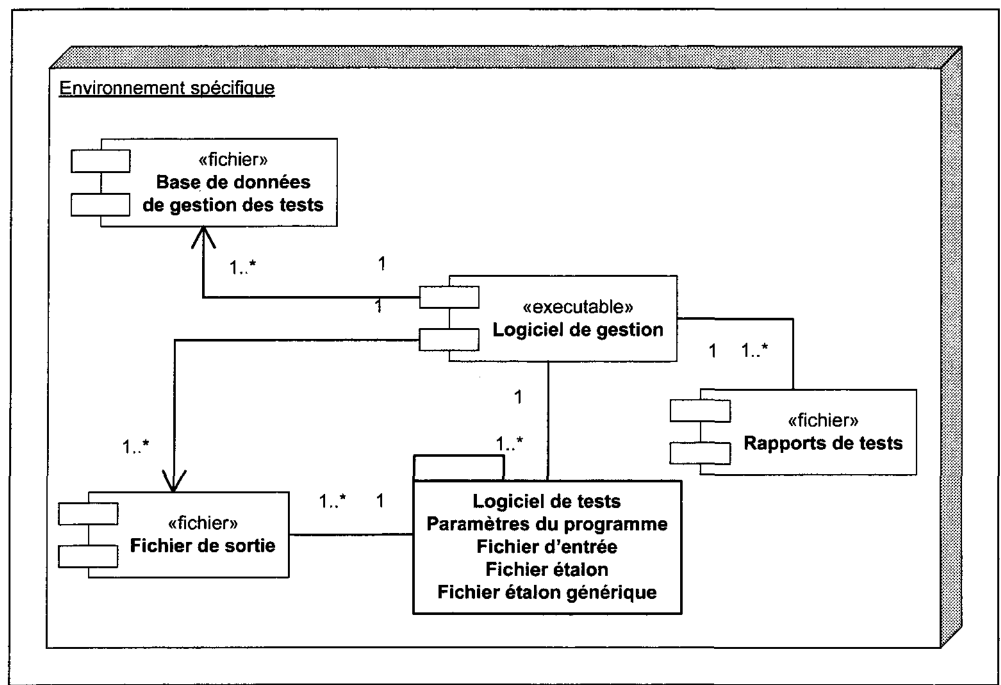
3.3.1 Séquence d'opérations

La séquence d'opérations de la méthode générique de gestion est basée sur une cohésion dans l'environnement (section 3.1.1 : Environnement) et sur la structure hiérarchique de l'élement *project* et la récursivité des éléments *groupSoftware* et *software*.

3.3.1.1 Dépendance du logiciel de gestion

Les dépendances du logiciel de gestion ont lieu dans l'environnement spécifique d'un logiciel de tests (le logiciel à tester). Ces dépendances sont présentées à la figure 24.

Figure 24 : Dépendance du logiciel de gestion



La première dépendance est celle entre le logiciel de gestion et la base de données de gestion des tests. Cette dernière est montrée ici sous forme de fichier. Dans cette dépendance, le logiciel de gestion interroge la base de données pour connaître quels sont les tests à effectuer. Le logiciel de gestion peut naviguer dans la base de données pour sélectionner les tests désirés. Ces tests sont constitués principalement de trois parties : le logiciel de tests, le fichier de données d'entrée et le fichier étalon. Sachant quels sont les tests à effectuer, le logiciel de gestion charge les données pour recevoir les fichiers qui sont nécessaires à l'exécution de ces tests. Ces fichiers à télécharger sont représentés ici sous forme de *package* car chaque test est unique.

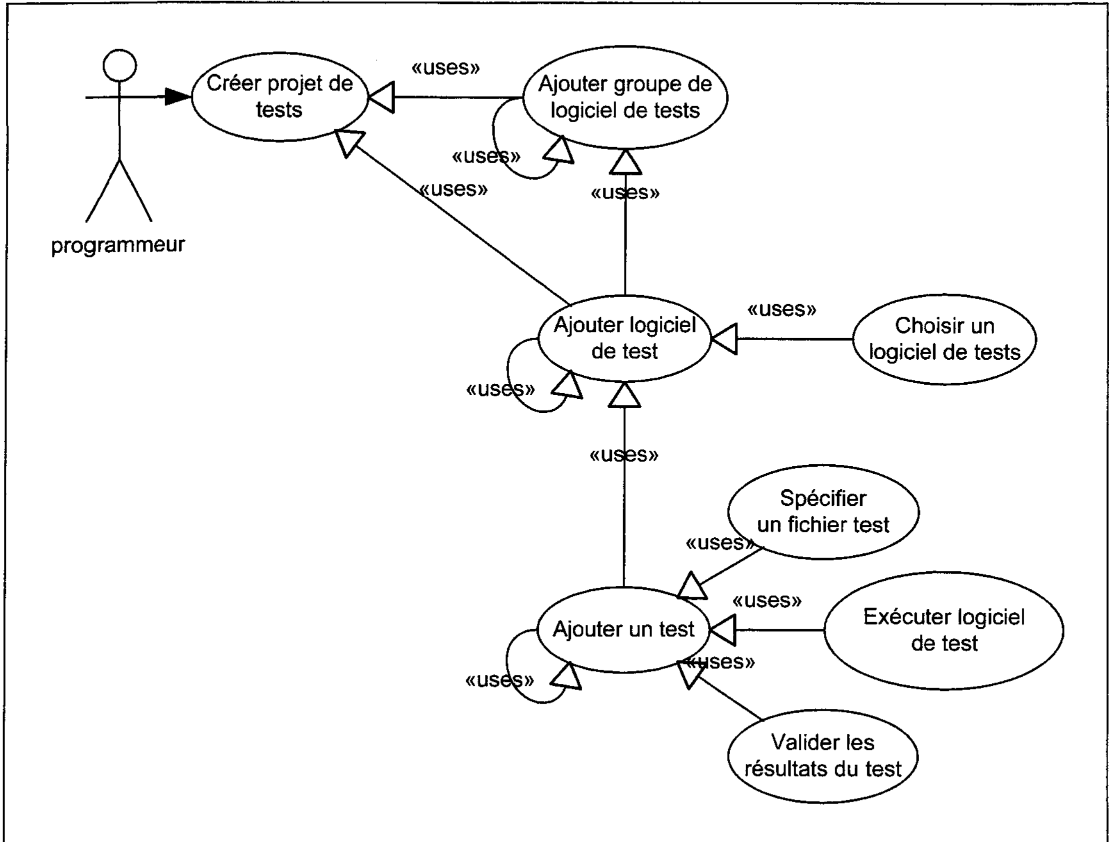
La seconde dépendance consiste à l'exécution du logiciel de tests avec le fichier de données d'entrée. Ceci génère un fichier de sortie.

La dernière dépendance consiste à comparer les fichiers de sortie et les fichiers étalons et à produire les rapports. Ces rapports sont présentés ici sous forme de composants fichiers pour illustrer que ceux-ci peuvent être transmis aux personnes qui en ont la responsabilité. Simultanément à sa transmission, une copie de chaque rapport est insérée dans la base de données des tests, ceci à des fins d'archivage.

3.3.1.2 Création d'un projet de tests

La création d'un projet de tests est la première étape vers la construction de tests de régression. La figure 25 montre les étapes à effectuer pour construire un tel projet.

Figure 25 : Création d'un projet de tests

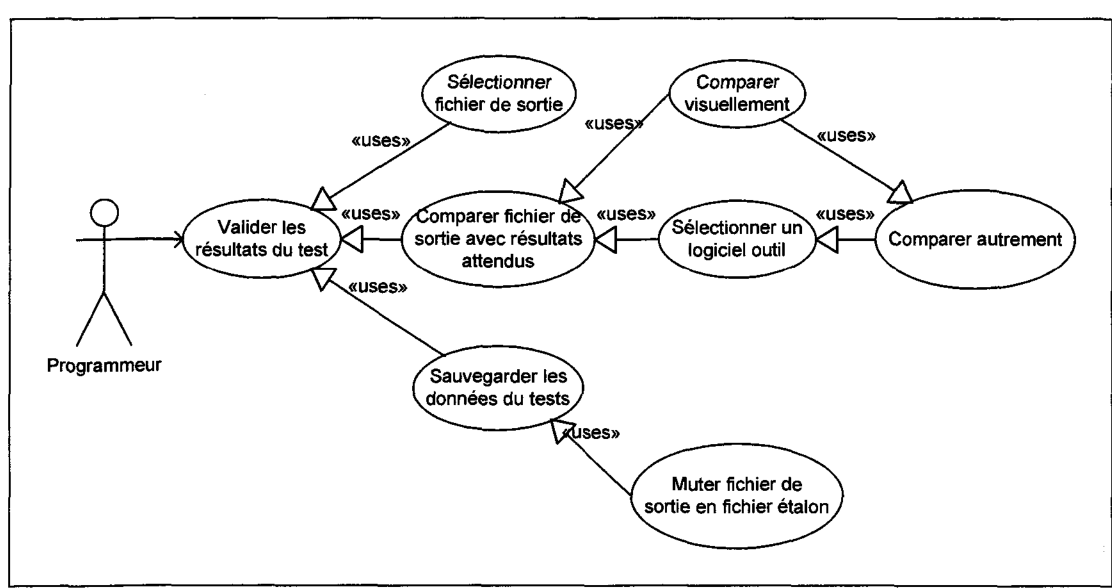


Tout d'abord, le programmeur crée un projet de tests. Ensuite, il peut y ajouter des logiciels à tester ou des groupes de logiciels à tester. Le lecteur se rappelle que les logiciels (élément *software*) sont pour un traitement séquentiel des tests tandis que les groupes de logiciels (élément *groupSoftware*) sont pour une exécution simultanée des logiciels de tests qu'il contient. Pour ajouter un logiciel de tests, il sélectionne celui-ci dans son répertoire de produits compilés et y ajoute les paramètres pertinents. Ici, le mot « paramètres » est pris au sens large du terme, c'est-à-dire qu'il inclut la possibilité d'activer des variables

d'environnement[24] ou des commandes a priori ou a posteriori du test[25]. Dans cette figure, certains modèles possèdent une récursivité. Ceci indique la possibilité d'une multiplicité.

Après avoir sélectionné le logiciel de tests, c'est-à-dire l'exécutable, le programmeur y ajoute un test. C'est lors de l'ajout de ce test que le programmeur spécifie le fichier de données qui sera utilisé. Puis, il exécute son test et valide les résultats du test. La figure 26 montre le modèle décrivant la validation des résultats.

Figure 26 : Valider les résultats du test



Pour valider les résultats du test, il faut tout d'abord sélectionner le fichier de sortie[26] puis en déterminer son extension. Le nom de l'extension est utilisé comme clef de recherche dans la table d'information sur les logiciels outils (section 3.3.1.4

---

[24] Toutefois, il est possible de placer les valeurs des variables d'environnement dans le fichier de tests. Dans ce cas, ce serait le logiciel de tests qui définirait les variables d'environnement à l'exécution du test.

[25] Ceci est particulièrement utile pour initialiser une base de données ou pour purger une base de données après un test.

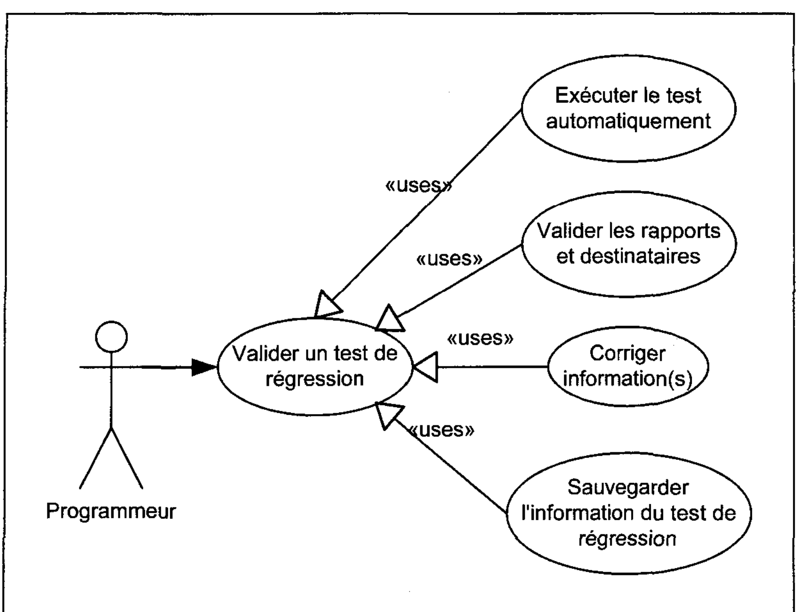
[26] Le même raisonnement est applicable s'il y a plusieurs fichiers de sortie.

: Logiciels Outils). Si la clef de recherche existe, alors le fichier de sortie est comparé autrement (section 3.3.1.3 : Comparer autrement). Si la clef de recherche n'est pas trouvée, alors le fichier de sortie est considéré comme un fichier texte et comparé visuellement avec les résultats attendus.

En pratique, il est suggéré de placer préalablement les résultats attendus dans un fichier et de comparer visuellement le fichier de sortie avec le fichier des résultats attendus. À ce stade, le jugement du programmeur est important, car même si les fichiers sont différents au niveau de l'apparence, ils peuvent être corrects au niveau du contenu. L'exemple le plus direct de ceci consiste en un fichier de sortie qui contient deux informations « Foo » et « Bar » sur une ligne, alors que le résultat attendu présente ces informations sur deux lignes, ou encore dans l'ordre « Bar » et « Foo ». Évidemment, l'interversion des termes n'est valide que si la position de l'un par rapport à l'autre est sans importance. Dans le cas où la comparaison visuelle n'est pas conforme, le programmeur doit investiguer la cause, faire la correction et recommencer le processus de préparation d'un test.

Suite à la création d'un projet de tests, il est important de valider les tests de régression qui viennent d'être insérés dans la base de données et qui contiennent toutes les informations pertinentes. La figure 27 décrit cette étape.

Figure 27 : Valider un test de régression

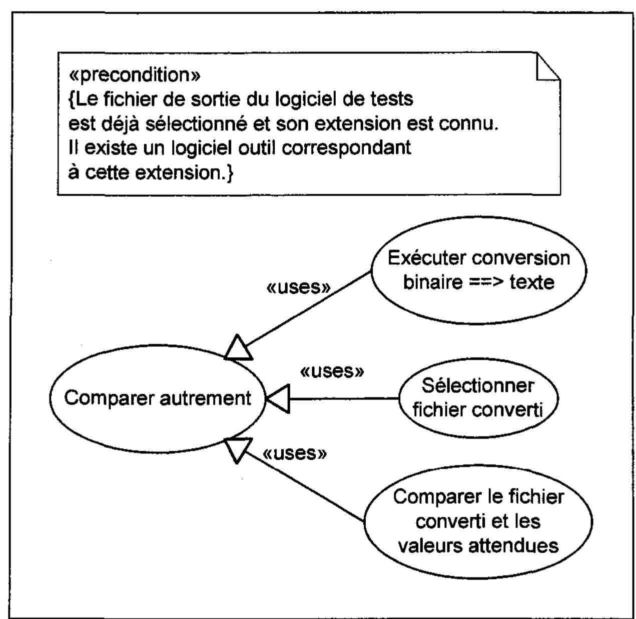


Pour valider le test de régression, le programmeur doit tout d'abord exécuter celui-ci. Cette exécution génère un fichier de sortie qui est automatiquement comparé avec le fichier étalon. Ces deux fichiers devraient correspondre. En cas de non-concordance, entre le fichier de sortie et le fichier étalon, celle-ci est affichée à l'écran. S'il y a concordance entre le fichier de sortie et le fichier étalon, les informations des destinataires des rapports de tests sont affichées. À noter qu'il est possible de fournir deux adresses de courriel pour le test. La première en cas de réussite, la seconde en cas d'échec du test. De plus, puisque l'exécution du test de régression est en mode construction du projet de tests, les rapports sont affichés à l'écran et non envoyés aux destinataires. Lorsque le test de régression est validé, le programmeur peut ajouter un nouveau test au projet.

3.3.1.3 Comparer autrement

Le modèle « Comparer autrement » traite les cas où le fichier de sortie est trop complexe pour être comparé visuellement, par exemple un fichier binaire. La figure 28 montre ce modèle.

Figure 28 : Comparer autrement

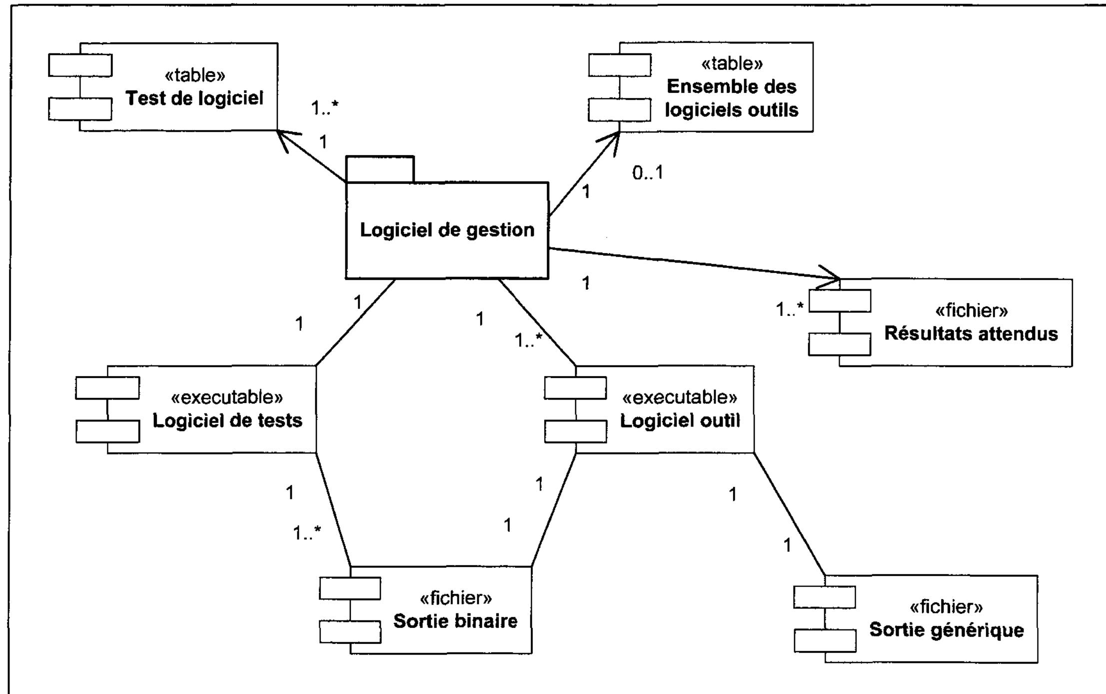


Pour « Comparer autrement » le fichier de sortie, il faut exécuter la conversion à l'aide du logiciel correspondant à l'extension de celui-ci. Ceci génère un fichier texte qui peut être validé visuellement par le programmeur.

3.3.1.4 Logiciels Outils

Les logiciels outils sont des logiciels qui traitent les fichiers de sortie d'un logiciel de tests. Ils sont utilisés pour transformer un fichier de sortie binaire en un format compréhensible par un humain et placer ce résultat dans un fichier. C'est ce dernier fichier qui sera comparé avec les résultats attendus. La figure 29 illustre les relations du logiciel de gestion avec les logiciels outils.

Figure 29 : Relations du logiciel de gestion avec les logiciels outils



À remarquer qu'il existe deux tables de données : la table des tests de logiciel et la table des outils logiciels disponibles. La première table contient les informations décrivant les tests. Lors de la construction d'un test, ces informations sont placées dans des variables du logiciel de gestion. Ce n'est que lorsque le test sera validé que celui-ci sera enregistré dans la table. La seconde table permet de trouver le logiciel requis pour traiter le fichier binaire généré par le logiciel de tests courant.

Lorsqu'un logiciel de tests génère un fichier binaire, le logiciel de gestion analyse le nom du fichier de sortie pour acquérir le nom de l'extension de celui-ci. Ensuite, une recherche est effectuée dans la table des outils selon le nom de l'extension du fichier de sortie qui vient d'être créé. Si un enregistrement ayant cette extension existe dans la table des outils, le logiciel associé est activé. L'outil ne prend qu'un seul paramètre : le nom du fichier à traiter. La sortie est effectuée dans un fichier générique ayant toujours le même nom et la même localisation

(informations qui sont dans la table des outils). Lorsque le programmeur sélectionnera son fichier des résultats attendus, il pourra comparer visuellement le fichier transformé et les résultats attendus.

Toutes les autres méthodes de comparaison seront traitées selon le modèle « Comparer autrement » (section 3.3.1.3 : Comparer autrement).

3.3.2 Comparaison fichiers de sortie/étalon

La comparaison entre un fichier de sortie et un fichier étalon est le fondement même des tests de régression. La méthode proposée permet une grande flexibilité quant à cette comparaison tant au niveau des comparaisons textuelles que binaires.

3.3.2.1 Fichier texte

La comparaison d'un fichier texte avec son fichier étalon peut s'effectuer selon deux modes : déterministe ou non déterministe.

3.3.2.1.1 Déterministe

La comparaison déterministe signifie que le fichier de sortie doit être identique au fichier étalon. Ce mode de comparaison est le plus contraignant des modes de comparaisons entre fichiers textes. Toutefois, l'utilisation des options textuelles de comparaison permet d'amoindrir les contraintes et de simplifier la tâche du programmeur — voir tableau 14 — tout en respectant l'esprit des comparaisons textuelles déterministes.

Selon les options du mode texte, la comparaison fichier de sortie/étalon peut s'effectuer en comparant un à un les jetons de ces fichiers. L'espacement entre deux jetons est alors considéré comme un seul caractère blanc peu importe la multiplicité de celui-ci.

Dans le tableau 14, la comparaison textuelle des fichiers « FOO1.txt » et « FOO2.txt » serait considérée comme vraie car les caractères blancs (entre `Foo` et `Bar`) de FOO2.txt sont considérés comme un seul blanc. Les caractères blancs sont : l'espace, la tabulation, le retour de chariot, le saut de ligne et la tabulation verticale.

Tableau 14 : Comparaison de jetons

Fichier	Ligne de texte
FOO1.txt	`Foo Bar`
FOO2.txt	`Foo          Bar`

Remarque : dans le cas où les deux fichiers doivent être identiques, il est possible de comparer des fichiers textes comme étant des fichiers binaires, tel que montré à la section 3.3.2.1.3 : Fichier binaire.

3.3.2.1.2 Non déterministe régulier

Un fichier de sortie est non déterministe si au moins une partie de celui-ci change de valeur de façon non prévisible à chaque exécution du test. Un fichier est non déterministe régulier si cette différence peut être localisée dans une zone, spatialement ou syntaxiquement, dans le fichier de sortie. Une zone spatialement localisée commence à une position X et se termine N octets plus loin, X et N étant connus et constants. Quant à une zone de localisation syntaxique, il s'agit d'une zone qui est délimitée par des identificateurs, c'est-à-dire que la donnée non déterministe est bornée par des caractères de reconnaissance. Par exemple, si les caractères `\[` et `\]` indiquent le début et la fin d'une zone non déterministe, la comparaison du fichier de sortie et du fichier étalon exclura la zone contenue dans ces bornes.

3.3.2.1.3 Non déterministe non régulier

Un fichier non déterministe non régulier est un fichier dans lequel la localisation d'au moins une information peut être différente à chaque exécution de tests, et ce, de façon non prévisible. Dans une première approximation, la méthode proposée ne gère pas ces cas de tests.

3.3.2.2 Fichier binaire

En mode binaire, la comparaison s'effectue bit à bit. L'exactitude de la comparaison du fichier étalon et de sortie est, de ce fait, triviale.

3.3.3 Rapport de tests

Le rapport de tests fournit une synthèse des résultats des tests de régression exécutés. Il est produit à chaque exécution des tests de régression et la transmission à la personne responsable est assurée par le moyen approprié : courriel, boîte de message au chargement du logiciel.

Si le test génère le résultat attendu, le rapport contient le message suivant :

Exemple 14 : Rapport de tests conformes

```
Logiciel: namesoftware_1
Test: nameTest_1 conforme
Entrée :->..\..\SoftwareTest\Fonctions\Test\in\file_1.txt
Sortie :->..\..\SoftwareTest\Fonctions\Test\out\file_1.out
Etalon :->..\..\SoftwareTest\Fonctions\Test\etalon\file_1.out
```

Le rapport contient le nom du logiciel, le nom du test ainsi que le verdict de la comparaison du fichier de sortie et de son étalon. Vient ensuite la localisation des fichiers d'entrée, de sortie et étalon.

Dans le cas où le test ne génère pas le résultat attendu, le rapport de tests contient le nom du logiciel, la position de l'erreur dans chaque fichier ainsi que le nom du test, tel que montré à l'exemple 15 :

Exemple 15 : Rapport de tests non-conforme

```
Logiciel: namesoftware_1
Test: nameTest_1 conforme

Erreur
Fichier de sortie: -
>..\..\SoftwareTest\Fonctions\Test\out\file_1.out<- ligne : 2
contient : ->3<-
Fichier étalon: -
>..\..\SoftwareTest\Fonctions\Test\etalon\file_1.out<- ligne
: 2 attendait: ->33<-
Arrêt de la comparaison des fichiers du test: nameTest_1
```

3.3.4 Code source, binaire et documentation

Le code source et l'ensemble des fichiers sont disponibles sous forme de fichier archive « .zip » à l'adresse HTTP://dimsboiv.uqac.ca/~sboivin/jfmichaud/MTest.zip. Un fichier « readme.txt » décrit la procédure d'utilisation. De plus, le logiciel Doxygen [65] a été utilisé pour générer la documentation du programmeur. Aussi, le présent document est accessible sous format « .pdf » dans l'archive située à l'adresse HTTP://dimsboiv.uqac.ca/~sboivin/jfmichaud/Memoire_JFMichaud.zip.

3.3.4.1 Fonctionnement

Le prototype développé utilise un fichier XML comme base de données pour l'information de gestion des tests. L'exécution des tests génère des rapports qui sont localisés dans le répertoire correspondant. Pour connaître le résultat d'un test, l'utilisateur doit ouvrir le rapport correspondant. Les rapports sont sauvegardés en mode texte et le nom du fichier est un assemblage de la date et de l'heure, chacun de ceux-ci en format international. Par exemple : « 20040810_150943.txt » est le rapport exécuté le 10 août 2004, à 15 heures 09 minutes 43 secondes. Ceci assure suffisamment l'unicité des fichiers de sortie pour le prototype jusqu'à ce que la base de données soit implantée.

3.3.4.2 Logiciel (SVC)

Le prototype de méthode générique de gestion des tests d'unités logiciel est compatible avec l'utilisation d'un système SVC. Pour effectuer les tests de régression ou ajouter de nouveaux tests, il suffit de télécharger la version courante du logiciel à tester à partir du serveur. Cette version inclut le logiciel de tests, les bibliothèques, les fichiers de tests, les fichiers étalons ainsi que les fichiers sources.

De la même façon, après l'ajout de code source, de tests ou de fichiers pertinents au projet, ceux-ci doivent être sécurisés en les téléchargeant sur le serveur. Dans le cas de la version prototype qui utilise le fichier XML, les rapports générés doivent aussi être ajoutés au serveur SVC.

3.3.4.3 Mode console

L'utilisation en mode console s'effectue par l'appel du logiciel et par le nom du fichier XML. L'exemple 16 montre la ligne de commande pour l'appel de `MTest`, version prototype. Le fichier XML est lu par une bibliothèque (`jdom.jar`) à partir du répertoire qui contient le script d'activation (`.\MTest\bat\do.bat`).

Exemple 16 : Activation mode console

```
java -cp ..\lib\jdom.jar;..\build\MTest.jar MTest ..\xml\mtest.xml
```

Avec les informations suivantes :

- `java -cp ..\lib\jdom.jar` : appelle le *Java Runtime* et spécifie le *classpath*,
- `..\build\MTest.jar` : localisation du logiciel MTest,
- `..\xml\mtest.xml` : localisation du fichier qui regroupe tous les tests.

Présentement, il n'y a pas d'option disponible pour tester seulement un projet, un groupe de logiciels, un logiciel ou un seul test, donc tous les tests du fichier XML sont réexécutés.

3.3.4.4 Construction de fichier XML de contrôle

Un fichier XML est présenté à l'annexe 4: Fichier de données XML. Pour l'utiliser, le programmeur doit écrire son projet de façon similaire à ce qui se trouve déjà dans ce fichier. En cas d'erreur de localisation de fichiers, MTest avisera l'usager par des messages d'erreur.

3.4 Expérimentation in situ

Cette section montre comment la méthode générique de gestion des tests est partie d'une simple méthode de comparaison visuelle pour devenir un modèle flexible et intégrable dans un logiciel de gestion. Pour ce faire, l'organisation du projet ainsi que les spécifications des modules utilisés seront décrites.

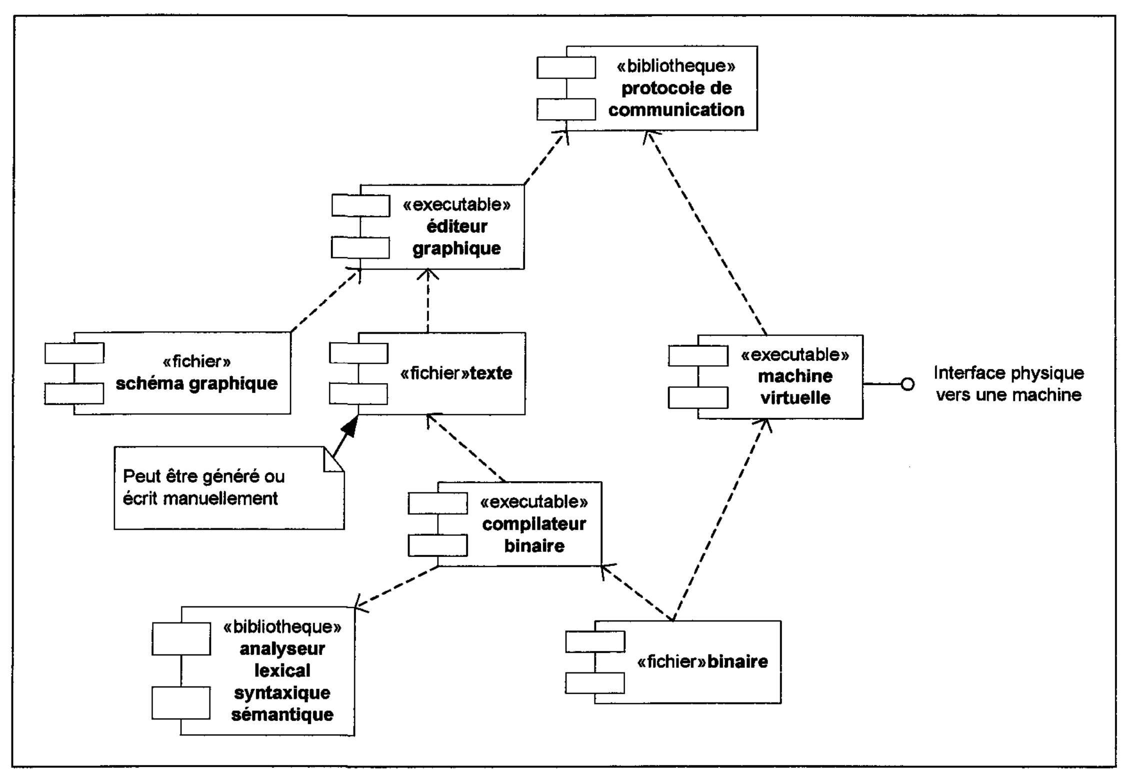
Avant de commencer, mentionnons que la méthode de développement lors de l'expérimentation était basée sur le *Feature Driven Development*. Au début du projet, le noyau de base des fonctionnalités à implémenter était connu et le développement était basé sur le paradigme du *Test Driven Development*. De ce fait, l'évolution du projet pouvait être comparée avec ces fonctionnalités. Aussi, le fondement de la méthode proposée a été utilisé lors du développement du langage Rodin, devenu par la suite K3, puis psC. L'évolution de cette méthode devint la méthode générique de gestion des tests d'unités logiciel.

Sans entrer dans les détails du langage psC ou de l'implémentation du projet, notons simplement que le but du langage était de faire, pour le domaine du contrôle de machine, ce que Java fait pour la programmation soit « construire une fois; exécuter partout ». Ceci a été rendu possible grâce à l'utilisation d'un générateur de code binaire et d'une machine virtuelle compilée pour le système d'opération ciblé.

3.4.1 Modèle du projet

Le modèle du projet représente l'organisation des modules tels qu'ils étaient à l'origine et non comme ils sont à ce jour. Les principaux composants du modèle présentés sont : éditeur graphique, compilateur binaire, analyseur lexical/syntaxique/sémantique, et machine virtuelle. La figure 30 montre l'agencement des différents modules.

Figure 30 : Modèle du projet



L'éditeur graphique est le module servant d'interface Humain-Machine. Selon le mode standard pour développer un programme de contrôle de machine, la première étape consiste à développer un schéma de contrôle puis à le stocker dans un fichier schéma graphique, pour usage ultérieur. Durant la création du schéma de contrôle, lorsqu'une étape de construction est complétée, l'utilisateur de l'éditeur graphique peut générer le fichier texte correspondant (la méthode de validation incrémentale est aussi applicable à la construction d'un schéma de

contrôle de machine). Celui-ci est alors validé par l'analyseur lexical/syntaxique/sémantique du compilateur. Si cette validation est positive, cela signifie que le schéma de contrôle satisfait les caractéristiques de la grammaire du langage et un fichier binaire est alors généré. Dans le cas contraire, le message d'erreur est affiché à l'usager. Pour terminer, le fichier binaire est transmis à la machine virtuelle qui l'exécute et contrôle des machines grâce à l'interface physique.

De plus, l'utilisateur peut simuler le fonctionnement de son schéma grâce à l'utilisation d'un protocole de communication. Dans ce cas, un fichier de tests doit être fourni avec le fichier binaire. De son côté, la machine virtuelle générera un fichier résultat qui sera comparé avec les résultats attendus. Cependant, puisque le fichier de tests est transmis par un protocole de communication, il est impossible de prédire le temps requis pour faire parvenir les données à la machine virtuelle. Il est alors impossible (de par la conception de la machine virtuelle) de prédire dans quel ordre seront les données générées dans le fichier de sortie.

D'autre part, une particularité de l'organisation de ce projet consiste en ce que le binaire produit par chaque module, ainsi que les fichiers d'en-tête et, le cas échéant, les fichiers reliés (par exemple : bibliothèque externe) sont regroupés dans un fichier archive (par exemple : « .zip ») propre à chaque module, avec son propre numéro de version. Lorsqu'un module a besoin d'un de ces fichiers archives, le programmeur télécharge du SVC l'archive adéquate. Un script, activé manuellement sur cette archive, place le contenu de celle-ci dans les répertoires appropriés pour la compilation du nouveau module. Cette méthode facilite la gestion des versions entre les différents modules et n'a pas besoin d'un logiciel de gestion de la configuration complexe ni de personnel dédié. La construction de ces archives est assurée par des scripts, déclenchés manuellement, avec comme paramètre le numéro de version. L'utilisation d'un SVC permet de séparer parfaitement les modules et de fournir à ceux-ci la possibilité d'avoir leur propre numéro de version. Le but étant de faciliter la cohésion des composants

développés indépendamment et de découpler ainsi l'avancement des travaux. Chaque module était habituellement la responsabilité d'un seul programmeur. Ici, le mot responsabilité est pris dans le sens de « le plus expert dans ce module », et non pas dans le sens « je suis le seul à le modifier ». Le principe de cette méthode était utilisé pour chaque module.

3.4.2 Tests de modules

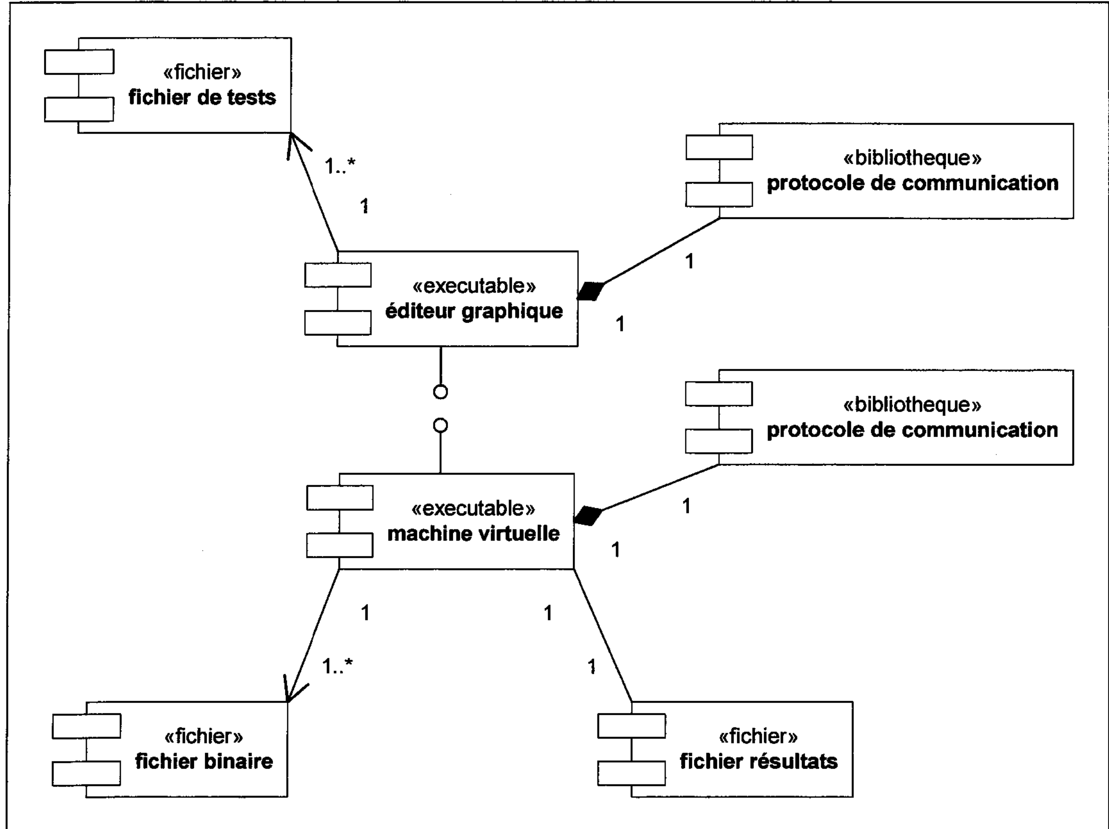
Les tests de modules sont les tests effectués par chaque programmeur sur son module et selon les spécifications de celui-ci. Les procédures de tests décrites sont celles qui ont été utilisées durant le développement. Chaque module possède des contraintes et caractéristiques qui, si elles sont regroupées ensemble, forment la spécification de la méthode générique de gestion des tests d'unités logiciel. Pour terminer, un résumé de ces contraintes est présenté et des méthodes qui permettent de les satisfaire sont proposées.

3.4.2.1 Interface graphique

La validation de l'interface de l'éditeur graphique était effectuée par mode d'essais et erreurs, sans aucune automatisation. Cette décision est valable dans le contexte où l'éditeur était considéré comme un prototype dont la tâche première est de générer correctement le fichier qui est transmis au module compilateur. Le protocole de communication utilisé a été testé manuellement par l'utilisation d'un port de communication entre deux ordinateurs.

La figure 31 montre cette particularité. Pour simplifier ce schéma, le fichier binaire a été posé comme étant préalablement créé.

Figure 31 : Test du protocole de communication



L'utilisateur de l'interface graphique peut, grâce au protocole de communication, contrôler la machine virtuelle et lui ordonner de charger et de démarrer le programme contenu dans le fichier binaire. L'utilisateur choisit alors un fichier dans la famille de ceux construits pour le fichier binaire. Sans entrer dans le domaine technique de la construction des fichiers de tests, mentionnons que pour un fichier binaire il peut y avoir un ou plusieurs fichiers de tests correspondants. Ici, l'éditeur graphique et la machine virtuelle communiquent au moyen d'une interface réseau, chacun ayant accès aux fonctions du protocole de communication. Ceci implique que, pour tester le protocole de communication, il est nécessaire d'avoir deux logiciels qui sont exécutés et qui peuvent s'échanger des messages par un port de communication. Ce facteur reviendra lors de l'élaboration des contraintes de tests.

Le troisième élément à tester est la génération du schéma de contrôle utilisé par le module compilateur. La procédure de tests consistait à faire valider celui-ci par l'analyseur. Si l'analyseur ne signalait pas d'erreur, le fichier texte était considéré comme correct. Dans le cas contraire, des démarches étaient entreprises pour trouver qui de l'éditeur graphique ou de l'analyseur était correct. Cette approche était suffisante car l'analyseur servait d'outil de validation puisqu'il était le mieux testé et servait de fondement au projet.

3.4.2.2 Compilateur

La première partie du fonctionnement du module compilateur consiste à utiliser les fonctionnalités de l'analyseur pour valider le fichier texte décrivant le schéma de contrôle de machine. La seconde partie consistait à tester la génération du fichier binaire qui était destiné à la machine virtuelle.

3.4.2.2.1 Analyseur

Dans le cas l'analyseur, la validation de l'implémentation du langage informatique a été effectuée grâce à la technique *parse-unparse,* c'est-à-dire : lire le fichier d'entrée, le traiter puis générer un fichier qui sera comparé sémantiquement avec le fichier d'entrée. La position des jetons dans le fichier pouvait être différente mais la sémantique devait être la même. Aussi, dans certains cas, de l'information de débogage des tables de symboles était insérée dans le fichier de sortie. Ceci est sans conséquence en regard de la correspondance des fichiers et s'applique particulièrement bien à l'implantation d'un langage informatique. Donc, une fois la méthode *parse-unparse* apprise, il est assez simple d'évoluer rapidement dans les tests à effectuer. Un des paradigmes de la méthode *Test Driven Development* est alors utilisé : *code a little, test a little*. La comparaison visuelle s'effectue aisément, à tout le moins avec très peu de cas de tests.

Le nombre de tests augmentant, il est devenu rapidement impraticable de revalider manuellement et visuellement les tests précédemment effectués. C'est à partir de ce moment que l'automatisation de la comparaison des tests a commencé. L'implantation en mode script a été développée et assimilée.

Pour valider la gestion correcte de la mémoire, une petite bibliothèque maison a été développée pour détecter les fuites de mémoire. Le résultat était placé dans un fichier texte. C'est à ce moment que l'idée d'avoir des « fichiers génériques étalons » est née.

3.4.2.2.2 Générateur de code binaire

Le générateur de code binaire génère un fichier binaire, ce qui rend caduque la validation visuelle de celui-ci. Pour valider l'implémentation du module compilateur, un logiciel a été écrit. Celui-ci traduisait le fichier binaire en texte pour que le programmeur puisse le valider. Si ce texte était différent, selon l'avis du programmeur, du fichier d'entrée, il y avait alors une erreur.

Des scripts d'automatisation ont été écrits. Ils permettaient de valider visuellement le résultat en quelques secondes. Cependant, cette automatisation n'est que pour un seul test, et non pour l'ensemble des tests du module compilateur. Il y a donc place à optimisation pour permettre d'effectuer des tests de régression. La solution devait impliquer l'utilisation d'un logiciel autre que celui du module compilateur. La notion d'outils de validation fournis par le programmeur commençait à prendre forme. Un second programmeur a utilisé le même processus de développement mais n'a pas optimisé sa méthode de travail, pour effectuer le même type de validation. Cette différence entre les méthodes de travail provoque des pertes de temps et de cohésion au sein de l'équipe de développement, ce qui est un facteur à corriger.

3.4.2.3 Machine virtuelle

Au bout de la chaîne de développement, le module machine virtuelle reçoit le fichier binaire, qui est le programme à exécuter pour contrôler les machines. Ce fichier est considéré comme valide, car il est le résultat du module compilateur. Il est très difficile de tester le fonctionnement de la machine virtuelle lorsqu'elle contrôle des machines. Cependant, il est possible, en mode simulation, de sauvegarder les résultats dans un fichier.

Cette technique est adéquate durant le développement de la machine virtuelle. Cependant, elle est difficilement applicable lorsque la machine virtuelle est en production car l'environnement cible de la machine virtuelle est les systèmes embarqués. Les principales raisons à cet état de fait sont :

- Demande de l'espace mémoire, pour stocker le fichier.
- Peut ralentir la vitesse de calcul du programme suffisamment pour que le contrôle de machine soit désynchronisé.

3.4.3 Contraintes et solutions

Les solutions proposées ici sont le résultat de plusieurs discussions avec les membres de l'équipe de développement. Des solutions pour permettre de tester efficacement ont été trouvées, et dans certains cas, implantées en tout ou en partie. Elles sont le fruit d'optimisation de méthode de travail et ont fait leurs preuves sur le terrain ou ont satisfait les exigences de l'expert du module. Le tableau 15 synthétise les contraintes de tests rencontrées dans les différents modules. Il contient aussi les solutions à ces contraintes et comment ces solutions sont présentes dans la méthode proposée.

Tableau 15 : Solutions proposées face aux contraintes de tests

Composants	Contraintes de tests	Problèmes	Solutions de test	Présence dans la méthode proposée
Interface graphique	Possède une interface graphique.	Nombreux éléments, méthodes et attributs. La machine d'états de ce composant est mal connue.	Tester les composants graphiques en envoyant des « événements » dans la pile du système d'opération.	Oui[27]. Un programme de tests peut charger un composant graphique et lui envoyer des événements ou préparer l'état du logiciel. Puis, il peut observer et mesurer le comportement du composant.
Compilateur	Génère un fichier binaire.	Utilisation d'un logiciel outil causant de nombreuses manipulations de fichiers pour afficher les résultats en mode texte.	Automatiser l'utilisation du logiciel outil, l'affichage et la comparaison bit à bit de fichiers (pour comparer avec le fichier étalon).	Oui[28]. Utilisation d'un logiciel outil pour transformer le fichier binaire en fichier texte et l'afficher automatiquement pour comparaison visuelle.
Compilateur	Nil.	Non-standardisation des méthodes de tests.	Utilisation d'un logiciel de gestion des tests d'unités logiciel.	Oui.
Compilateur	Nombreux tests effectués par incréments de fonctionnalités.	Nombreux fichiers de tests.	Idem.	Oui.

[27] L'utilisation de bêta-testeurs peut être plus efficace, pour autant que les bogues rapportés soient rapidement corrigés. Il est onéreux d'automatiser les tests de l'interface graphique.

[28] Des spécifications sur la construction de cet outil doivent être fournies. Par exemple : dans quel répertoire doit être généré le fichier de sortie.

Tableau 15 : Solutions proposées face aux contraintes de tests (suite)

Composants	Contraintes de tests	Problèmes	Solutions de test	Présence dans la méthode proposée
Analyseur	Construction incrémentale du fichier de tests.	Difficulté de retrouver la zone d'intérêt dans le fichier de sortie si celui-ci devient grand.	Mettre des codes de couleurs et des fonctions de recherches des différences dans les fichiers de sortie et étalon.	Oui. (Développement futur d'une interface visuelle).
Analyseur	Construction incrémentale du fichier étalon.	Réorganisation visuelle du fichier de sortie pour simplifier le repérage visuel des jetons.	Comparaison jeton à jeton des fichiers de sortie et étalon.	Oui. (Développement futur) : mutation automatique du nouveau fichier de sortie validé en fichier étalon.
Machine virtuelle	Simulation du fonctionnement événementiel.	Des données peuvent s'intervertir dans le fichier résultats.	Outil de validation du fichier de sortie.	Oui.
Machine virtuelle	Simulation du fonctionnement événementiel.	Des données peuvent changer dans le fichier de sortie, sans changer la validité du fichier de sortie.	Utilisation de zone d'exclusion de comparaison entre les fichiers de sortie et étalon.	Oui.

À ces contraintes et solutions s'ajoute le fait que le processus de validation est maintenant standardisé, ce dernier facteur simplifiant la formation du personnel. De même, la solution proposée permet de simplifier le travail du programmeur et de standardiser la méthode de développement et de tests parmi l'équipe de développement.

Les solutions proposées entrent dans la catégorie « *test driven* » et offrent une très grande flexibilité avec de nombreux avantages :

1. Incrémentation des tests appliqués sans recompilation du code source;
2. Hiérarchie de tests;
3. Applicable à tous les langages de programmation;
4. Applicable aux bases de données;
5. Automatisation des tests de régression;
6. Grande flexibilité des conditions de tests;
7. Applicable aux tests en boîte noire;
8. Applicable aux tests en boîte blanche;
9. Comportement similaire pour les versions compilées en mode *debug** ou en mode *release**.

Parmi les désavantages se trouvent :

1. Applicable seulement à des tests d'unités logiciel et non à un logiciel assemblé (à moins que le logiciel assemblé n'inclue les fonctionnalités nécessaires);
2. Construction des projets et des makefile* plus complexes;
3. Gestion plus complexe de la configuration des composants avec les versions des fichiers de tests.

3.5 Discussion

La méthode de gestion des tests d'unités logiciels présentée dans ce mémoire est applicable dans plusieurs environnements et, plus spécifiquement, à plusieurs langages de programmation. Les éléments qui la composent offrent une grande flexibilité d'utilisation dans divers systèmes d'opération. De plus, elle offre des fonctionnalités qui lui permettent d'effectuer des tests de régression produisant des fichiers texte non déterministes réguliers. Elle est aussi applicable aux cas particuliers où le fichier de sortie contient des zones ciblées de variabilité, contrairement aux logiciels de la famille xUnit qui fonctionnent de façon absolument déterministe au niveau de la validation. La méthode proposée offre une solution au cas particulier des modules qui génèrent des fichiers binaires, par l'utilisation de logiciel outils, et elle peut être appliquée aux cas où deux logiciels ou plus doivent être exécutés concurremment.

Aussi, puisque les informations des projets, des logiciels et des tests effectués (incluant les fichiers étalons) sont dans une base de données, il est possible de sélectionner l'ensemble de tous les projets, un seul projet ou même un seul logiciel pour effectuer les tests de régression. Cette tâche peut évidemment être effectuée en fonction de divers systèmes d'opération.

Le fondement de la méthode a été utilisé pendant 18 mois sur un projet. L'équipe était formée de cinq programmeurs et d'un gestionnaire. Le projet développé comportait environ 60 000 lignes de code, dont environ 40 000 en environnement Windows (C++) et 20 000 en environnement QNX (ANSI C). Cette utilisation a mené au développement de la méthode proposée. L'expérience a montré que la méthode est facile à apprendre et que l'équipe de développement en voit rapidement l'intérêt.

En outre, si une entreprise utilise un langage unique dans un système d'opération unique et ne fabrique pas de logiciels exécutés simultanément, ni ne se sert de fichier binaire intermédiaire, il est possible qu'un logiciel de la famille xUnit puisse être utilisé efficacement. Cependant, la contrainte de rester dans un cadre strictement déterministe rend l'utilisation des xUnit inférieure à la méthode proposée.

Quoique la méthode proposée soit capable d'automatiser le comportement des interfaces informatiques par l'utilisation de programmes de tests et de fichiers de données de tests qui contiendraient la liste des événements à produire, il est incertain qu'elle puisse le faire de façon efficace et économique pour la petite entreprise.

Si un projet exploite le paradigme Modèle—Vue—Controlleur, la méthode proposée est adéquate pour tester les modules Modèle et Controlleur. Quant au module Vue, il semble plus avantageux que le programmeur exécute les tests de fonctionnement positif, et de laisser le soin à des bêta-testeurs de valider le fonctionnement négatif de l'interface.

Pour terminer, voici les avantages que l'on retire de la méthode proposée par opposition à l'utilisation des xUnit.

1. Aucun *overhead* dans le produit compilé;
2. Simplicité de la conception;
3. Apprentissage unique de la méthode;
4. Applicable à divers système d'opération;
5. Permet une complexité aussi grande/petite que désirée du fichier de sortie;
6. Regroupe les tests visuellement à un endroit (fichier d'entrée);
7. Niveau multiple de rapports de tests;
8. Applicable aux fichiers binaires (image, film, son, fichier crypté);
9. Permet un non-déterminisme dans les fichiers de sortie.

D'un autre côté, la méthode proposée possède certains désavantages qui sont :

1. Outils non matures;
2. Petit nombre d'utilisateurs;
3. Probable difficulté pour les tests d'interface Humain-Machine.

3.6 Développements futurs

Les développements futurs sont potentiellement nombreux mais visent tous à une validation in situ de la méthode proposée. Cependant, il est prudent de minimiser l'information que le programmeur aura à gérer.

L'utilisation d'une base de données pour stocker les rapports permet de gérer la transmission des rapports aux personnes responsables par le moyen le plus adapté à l'organisation : courriel, messagerie interne ou autre. De même, la base de données permettrait au gestionnaire d'assurance-qualité de faire l'assemblage des informations contenues dans le rapport.

Les rapports de tests sont construits de façon à garder précisément les traces des tests effectués en enregistrant, minimalement, les informations suivantes :

- Le nom du projet;
- Le nom du logiciel;
- Le nom du test;
- La date de l'exécution du test;
- Le verdict de la comparaison fichier de sortie/étalon;
- Le cas échéant, les localisations des différences entre le fichier de sortie et le fichier étalon.

Dans la version prototype, tous les rapports de tests sont inclus dans un fichier texte. Le nom de ce fichier est selon le format « aaaammjj_hhmmss.txt ». En supposant que l'exécution de ce test et la comparaison du fichier de sortie avec son étalon prennent une seconde ou plus, le nom du fichier est unique et conséquemment, il n'y aura pas de collision entre les noms des rapports. Dans le cas où cette supposition est erronée, des conditions temporelles doivent être implémentées dans le logiciel de gestion.

Le logiciel de gestion des tests peut être implémenté sous forme d'applet Java. Celui-ci sera alors exécuté dans un fureteur et accédera localement aux fichiers requis. Il sera toutefois indispensable de lui accorder les permissions nécessaires selon les systèmes d'opérations.

L'amélioration la plus utile et importante, advenant que le logiciel soit développé selon le modèle *Open Source*, est la création d'une interface graphique. Celle-ci permettrait de simplifier la tâche de coordonner les projets, les logiciels, les groupes de logiciels, les fichiers de tests, les fichiers de sorties, les fichiers étalons, les rapports de tests, les variables d'environnements, les zones d'exclusion de tests, et les listes d'outils configurables par l'usager.

CONCLUSION

Dans ce mémoire de maîtrise, nous avons présenté une méthode générique de gestion des tests d'unités logiciel. Celle-ci a été conçue pour s'appliquer dans les petites entreprises d'informatique (2 à 10 programmeurs) et à divers types de livrables informatiques (bibliothèque, DLL ou exécutable) ou de langages de programmation et dans divers systèmes d'opération. De même, la structure de répertoire nécessaire pour le fonctionnement de la méthode est archivée dans un logiciel de SVC, ce qui simplifie la gestion des versions des tests.

Une des forces principales de la méthode proposée est que le programmeur apprend une seule architecture de conception et l'utilise dans tous ses langages de programmation. Dans le cas d'interfaces graphiques, la méthode favorise une utilisation hybride de méthodes de tests : des tests a priori pour les unités de calcul et des tests a posteriori pour les interfaces graphiques.

Pour ce mémoire, une situation réelle de la petite entreprise a été utilisée. Des solutions ont été présentées pour chacun de ces problèmes. Toutefois, pour valider la méthode proposée, une implantation in situ devra être réalisée. De plus, il est plausible que la méthode proposée soit applicable à de plus grandes entreprises.

BIBLIOGRAPHIE

[1] Moncur, M. (2005). *The Quotations Pages*. Accédé avril, 2005, Site http://www.quotationspage.com/quote/879.html

[2] Hanson, R. J. (1996, 1996//). *Test strategies/innovations for the 90s*. Papier présenté à Proceedings of Nepcon East, 10-13 June 1996, Boston, MA, USA.

[3] Kone, O., & Castanet, R. (2000). Test generation for interworking systems. *Computer Communications, 23*(7), 642-652.

[4] Shemesh, Y. (2002 avril). *Tools Breakout*. Accédé Novembre, 2004, Site shemesh.larc.nasa.gov/foot/crab/Breakout-Tools.ppt

[5] Boehm, B. (2002). Get ready for agile methods, with care. *Computer, 35*(1), 64-69.

[6] Nerur, S., Mahapatra, R., & Mangalaraj, G. (2005). Challenges of migrating to agile methodologies *Commun. ACM 48* (5), 72-78

[7] LeJeune, N. F. (2006). Teaching software engineering practices with Extreme Programming *J. Comput. Small Coll. , 21* (3), 107-117

[8] Marick, S. (1999, 2000-03-28). *New Models for Test Development*. Accédé décembre, 2004, Site http://www.testing.com/writings/new-models.pdf

[9] Boehm, B. W. (1988). A spiral model of software development and enhancement. *Computer, 21*(5), 61-72.

[10] Beedle, M., & al. (2001). *Manifesto for Agile Software Development*. Accédé mai, 2004, Site http://agilemanifesto.org/

[11] Berard, E. V. (2003). *Misconceptions of the Agile zealots*. Accédé août, 2004, Site http://www.toa.com/pub/Misconceptions.zip

[12] Abrahamsson, P., Warsta, J., Siponen, M. T., & Ronkainen, J. (2003). New directions on agile methods: a comparative analysis. In *Proceedings of the 25th International Conference on Software Engineering* (pp. 244-254). Portland, Oregon: IEEE Computer Society.

[13] Pressman, R. S. (1992). *Software engineering a practitioner's approach* (3rd. ed.). New York: McGraw-Hill.

[14] Beck, K. (2003). *Test Driven Development: By Example* (1st ed.). Boston: Addison-Wesley.

[15] Beck, K. (2000). *Extreme programming explained embrace change*. Boston: Addison-Wesley.

[16] Janzen, D., & Saiedian, H. (2005). Test-driven development concepts, taxonomy, and future direction. *Computer, 38*(9), 43-50.

[17] AdaptionSoft.*Test-Driven Development : Way Fewer Bugs*. Accédé juin, 2004, Site: http://www.adaptionsoft.com/tdd.html

[18] Nebulon_Pty_Ltd. (2002, 2004). *Feature Driven Development*. Accédé Septembre, 2004, Site http://www.featuredrivendevelopment.com

[19] Machiavelli, N. (1992). *Le Prince*. Paris: Garnier-Flammarion.

[20] Beck, K. (1999, 2004-02-28). *Extreme Programming : a gentle introduction*. Accédé mai, 2004, Site http://www.extremeprogramming.org/

[21] Jeffries, R. E. (2001-2005). *An Agile Software Development Resource*. Accédé mai, 2004, Site http://www.xprogramming.com

[22] Brewer, J. (2001). *Extreme Programming FAQ*. Accédé décembre, 2004, Site http://www.jera.com/techinfo/xpfaq.html

[23] AdaptionSoft.*Test-Driven Development : Way Fewer Bugs*. Accédé juin, 2004, Site http://www.adaptionsoft.com/tdd.html

[24] Cockburn, A., & Williams, L. (2001). The costs and benefits of pair programming. In *Extreme programming examined* (pp. 223-243): Addison-Wesley Longman Publishing Co., Inc.

[25] van Deursen, A., van den Bergh, A., Kok, G., & Moonen, L. (XP2001). *Refactoring test code*. Papier présenté à 2nd International Conference on eXtreme Programming and Flexible Proceses in Software Engineering. pages 92-95, 2001. Aussi disponible: http://www.cwi.nl/~leon/papers/xp2001/xp2001.pdf.

[26] Raymond, E. S. (2000). *The Cathedral and the Bazaar*. Accédé juin, 2004, Site http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/index.html

[27] Raymond, E. S. (2000). *Necessary Preconditions for the Bazaar Style*. Accédé juin, 2004, Site http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/ar01s10.html

[28] Khramtchenko, S.*Comparing eXtreme Programming and Feature Driven Development in academic and regulated environments*. Accédé août, 2004, Site http://www.featuredrivendevelopment.com/files/FDD_vs_XP.pdf

[29] Hoffman, D. (1998). *A taxonomy for Test Oracles*. Accédé mai, 2004, Site http://www.softwarequalitymethods.com/Papers/OracleTax.pdf

[30] Hoffman, D. (1999, mars/avril). Heuristic Test Oracles. *STQE Magazine. Aussi disponible: http://www.softwarequalitymethods.com/Papers/STQE%20Heuristic.pdf*.

[31] Pan, J. (1999 printemps). *Software testing*. Accédé mai, 2004, Site http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/#taxonomy

[32] Berard, E. V. (1994). *Issues in the testing of object-oriented software*. Papier présenté à Electro/94 International. Conference Proceedings. Combined Volumes. Accessible: http://www.toa.com/pub/oo_testing_article.txt.

[33] Myers, G. J. (1979). *The art of software testing*. New York ; Toronto: J. Wiley.

[34] IEEE standard glossary of software engineering terminology. (1990). *IEEE Std 610.12-1990.*

[35] Hetzel, W. C. (1988). *The complete guide to software testing* (2nd ed.). New York, N.Y.: J. Wiley and Sons.

[36] Mockus, A., Fielding, R. T., & Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol., 11*(3), 309-346.

[37] Open Source Technology Group. (2001, 2005). *Logiciel de gestion de projet Open Source: SourceForge*. Accédé mai, 2004, Site http://sourceforge.net/

[38] Graves, T. L., Harrold, M. J., Kim, J.-M., Porter, A., & Rothermel, G. (2001). An empirical study of regression test selection techniques *ACM Trans. Softw. Eng. Methodol. , 10* (2), 184-208

[39] Dijkstra, E. W. (1969). *Notes On Structured Programming*: Technological University Eindhoven. Disponible: http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.

[40] Software Quality Engineering. Accédé mai, 2004, Site http://www.stickyminds.com

[41] Software Research inc. (1994, 2005). *Quality techniques Newsletters QTRN.* Accédé avril, 2004, Site http://www.soft.com/News/TTN-Online/

[42] *Groupe de nouvelles.* Accédé mai, 2005, Site http://groups.google.ca/group/comp.software.testing

[43] Internet FAQ archive.*comp.software.testing Frequently Asked Questions (FAQ).* Accédé avril, 2005, Site http://www.faqs.org/faqs/software-eng/testing-faq/

[44] Cumming, P. G. (11 août, 2004). *Catching Integer Overflow.* Accédé avril, 2005, Site http://teaching.idallen.com/c_programming/catchingIntegerOverflow.html

[45] IBM Corporation.*General Decimal Arithmetic.* Accédé mai, 2005, Site http://www2.hursley.ibm.com/decimal/

[46] Bieman, J. M., Dreilinger, D., & Lin, L. (1996). *Using fault injection to increase software test coverage.* Papier présenté à Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on. Disponible http://www.cs.colostate.edu/~bieman/Pubs/issre96preprint.pdf.

[47] Weast, R. C., Lide, D. R., Astle, M. J., & Beyer, W. H. (1980). *CRC handbook of chemistry and physics a ready-reference book of chemical and physical data* (61th ed.). Boca Raton, Flor.: CRC Press.

[48] Nyman, N., (Microsoft Corporation). (2004). *In Defense of Monkey Testing.* Accédé août, 2004, Site http://www.softtest.org/sigs/material/nnyman2.htm

[49] Edvardsson, J. (1999 oct.). *A survey on automatic test data generation.* Papier présenté à Second Conference on Computer Science and Engineering, Linköping.

[50] Gotlieb, A., Botella, B., & Rueher, M. (2000). A CLP Framework for Computing Structural Test Data (Disponible http://www.essi.fr/~rueher/Publis/cl2000.pdf). In *Proceedings of the First International Conference on Computational Logic* (pp. 399-413): Springer-Verlag.

[51] Michael, C. C., McGraw, G., & Schatz, M. A. (2001). Generating Software Test Data by Evolution. Disponible http://citeseer.ist.psu.edu/mcgraw97generating.html. *IEEE Trans. Softw. Eng., 27*(12), 1085-1110.

[52] Larman, C. (1998). *Applying UML and patterns an introduction to object-oriented analysis and design.* Upper Saddle River, N.J.: Prentice-Hall.

[53] Clifton, M. (2004 janvier, 2004 mars). *Advanced Unit Test, Part V - Unit Test Patterns*
(An Introduction To The Concept Of Unit Test Patterns). Accédé juin, 2004, Site http://www.codeproject.com/gen/design/autp5.asp

[54] Médinas, R., & Novelle, T.*La programmation pilotée par les tests.* Accédé juillet, 2004, Site http://www.design-up.com/methodes/testsunitaires/index.html

[55] (2003, 2005/06/01). Site http://www.mockobjects.com/FrontPage.html

[56] Reasoning Inc.*Automated Software Inspection : A New Approach to Increased Software Quality and Productivity.* Accédé mai, 2004, Site http://www.reasoning.com/pdf/ASI.pdf

[57] Krasner, G. E., & Pope, S. T. (1988). A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System (Disponible: http://citeseer.ist.psu.edu/krasner88description.html). *J. Object Oriented Program.*, *1*(3), 26-49.

[58] Linz, T., & Daigl, M.*How to Automate Testing of Graphical User Interfaces*. Accédé juin, 2004, Site http://www.imbus.de/forschung/pie24306/gui/aquis-full_paper-1.3.shtml

[59] Beck, K.*Simple Smalltalk Testing : With Patterns*. Accédé juillet, 2004, Site http://www.armaties.com/testfram.htm

[60] Cunningham, W.*Testing Framework*. Accédé juin, 2004, Site http://c2.com/cgi/wiki?TestingFramework

[61] Free Software Foundation inc. (1996, 2005/06/07). *GNU General Public License*. Accédé avril, 2005, Site http://www.gnu.org/copyleft/gpl.html

[62] *JUnit Test Infected: Programmers Love Writing Tests*. Accédé avril, 2005, Site http://junit.sourceforge.net/doc/testinfected/testing.htm

[63] Morin, L. (2000). Software management Reference Card: Laboratoire de développement de logiciel, Université du Québec à Chicoutimi.

[64] Saint-Exupéry, A. d. (1957). *Vol de nuit*. Paris: Gallimard.

[65] Van Heesch, D. (1997, 2005). *Doxygen*. Accédé juillet, 2004, Site http://www.stack.nl/~dimitri/doxygen/

ANNEXE

1 : Glossaire

Bloc de code : élément de code informatique qui débute à un endroit et possède un seul début et une seule fin.

Code spaghetti : code informatique dont le déroulement est inutilement complexe, habituellement le résultat d'une mauvaise conception ou d'une implémentation déficiente.

Composant : élément faisant partie d'un produit. Par exemple un fichier « .dll ». Est composé d'une ou plusieurs unités.

Debug : compilation en mode *debug* : version du logiciel construite avec des options de compilations servant à localiser, le cas échéant, les bogues.

Debugging : action de trouver la cause exacte d'une erreur et d'en enlever la cause.

Fichier étalon : contient la réponse attendue pour les données d'entrée fournies. Ce fichier est comparé avec le fichier de sortie. La littérature anglaise utilise le terme *Oracle.*

Graphe de flot de contrôle : représentation logique d'un logiciel sous forme de graphe. Les nœuds sont les blocs de code et les arêtes sont les conditions de branchement.

Implémenter : réaliser la phase finale d'élaboration d'un système informatique, afin de le rendre fonctionnel[1].

Implémentation : opération qui consiste à réaliser la phase finale d'élaboration d'un système, afin de le rendre fonctionnel[2].

Makefile : fichier utilisé pour la compilation et l'édition des liens d'un projet logiciel.

---

[1]Office québécois de la langue française : HTTP : //www.oqlf.gouv.qc.ca/ressources/gdt.html
[2]Idem

Produit : Est le résultat de l'accomplissement d'un projet. Selon les méthodes employées, ce produit peut être une bibliothèque dynamique/statique ou un logiciel exécutable. Un produit peut être fait d'un ou plusieurs composants.

Release : compilation en mode *release* : version du logiciel construite avec des options de compilation pour optimiser en vitesse ou en grosseur le logiciel.

SVC (Système de Version Concourante) : base de données utilisée comme entrepôt pour sécuriser le code source de plusieurs logiciels et gérer les variantes et les versions de ceux-ci.

ANNEXE

2 : Logiciel de tests

```
// Progtest.cpp : Programme de test pour les opérations mathématique de base.

/* Fichier de données pour le module fonction: (traitement des opérations mathématique de
base)

  Date         par              raison      Version    Descriptif
  2004-08-03   JnFrs Michaud    Création      0.1      Création
  2004-08-09   JnFrs Michaud    Modif         0.1      Ajout des données de soustraction



Spécification du fichier de tests:

Le fichier de test ne doit pas avoir de ligne vide.


Format d'une ligne:
(Les donnée sont séparées par un/des espaces ou par une tabulation.

 #   Description
 1   Signe de l'opérateur
 2   Premier opérante
 3   Deuxième opérante

Exemple:

 + 2  2
 + 24 24

Algorithme:

 Note: il est possible d'introduire  le champ #4: résultat attendu,
       Le programme de test comparerait alors le résultat obtenu avec le résultat
       attendu, améliorant ainsi la vitesse de développement.

Le fichier de tests aurait alors le format suivant:

+  2  2  4
+  24 24 48


*************************************************************************
NOTE:

Ce code source de programme de tests fonctionne avec le fichier à
3 arguments.  Le résultat va dans le fichier de sortie.
*************************************************************************
*/

#include <string>
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <math.h>

#include "fonction.h"

using namespace std;
```

```
int main (int argc, char *argv[])
{

  string STRin    = "";       // Input file
  string STRout   = "";       // Output file
  string STRerror = "";       // Output file of error


  printf("Nombre argument: argc: %i\n", argc);
  if ( argc != 3 )
  {
    printf("Nombre de paramètres invalid!");
    printf("\nParamètres: fichier d'entrée, fichier de sortie.");
    printf("\nExemple: FOOBAR ../Test/in/FichierIn.txt ../Test/out/File_1.out");
    return 1;
  }

  STRin.append(argv[1]);              // remplit les strings
  STRout.append(argv[2]);

  ifstream ifSource(argv[1]);         // Ouverture des fichires
  ofstream ofOutput(argv[2]);         // Pour simplicité, la protection de
                                      // l'ouverture est retiré.
  string sOper ;

  int iNum1;
  int iNum2;


  while( ifSource >> sOper) // Discards newline char
  {
    try
    {
      ifSource >> iNum1;
      ifSource >> iNum2;

      int iResult ;
      double dResult;

      switch (sOper.at(0))
      {
      case '+':
        {
          iResult = Somme(iNum1, iNum2);
        }
        break;

      case '*':
        {
          iResult = Multiplication(iNum1, iNum2);

        }
        break;

      case '-':
        {
          iResult = Soustraction(iNum1, iNum2);

        }
        break;

      case '/':
        {
          dResult = Division(iNum1, iNum2);
```

```
            }
            break;

          default:
            {
              ofOutput << "Operateur->"<<sOper.c_str() << "<-Non valide" << endl;
            }
            break;

          }
          ofOutput << "iNum1" <<sOper.c_str() << "iNum2" << endl;
          ofOutput << iNum1 <<sOper.c_str() << iNum2<< "= ";

          if (strcmp (sOper.c_str() , "/") == 0)
          {// is division
            ofOutput << dResult << endl;

          }
          else
          {//is others
            ofOutput << iResult << endl;
          }
        }
        catch (...)
        {// ceci attapera le cas de la division par zéro.

          string str = "";

          str +=  "Catched exception\n";
          str += "iNum1->"+sOper+"<-iNum2\n";

          ofOutput << str;
          ofOutput << iNum1 <<sOper.c_str() << iNum2<< " =  EXCEPTION" << endl;
          ofOutput << "++++++++++++++++++++++++\n";
        }
      }
      ///:~
      printf("\n* * * End of testing file: ");

      ifSource.close();

      ofOutput.close();

      return EXIT_SUCCESS;

    }
```

ANNEXE

3 : Fichier script

```
echo off
rem FILE: testall.bat
rem this file will generate all schemeX.out and unparse
rem 21 mai 1999: unification des test de 16 et 32 bits, incluant les nombres.
rem              Les erreurs de bornes sur les nombres entiers et reels sont désirées.
rem
rem The file AllError is not yet included in the present test.  It must be done manually.
rem
rem 25 mai 2000: doing a dynamic path for testing.: JnFrs
rem Note:  Ne pas placer les chemins des path dans le fichiers ci-dessous,
rem        Ils sont dans le fichier Rodin3.cpp

rem
cls

Echo.
rem Executing test in batch.

cd..
cd distrib
if not exist rodin3.exe goto NoRodin3

Echo Working in Scheme*, in 16 bits
rodin3 scheme0 16>..\out\t0_16.txt>nul
rodin3 scheme1 16>..\out\t1_16.txt>nul
rodin3 scheme2 16>..\out\t2_16.txt>nul
rodin3 scheme3 16>..\out\t3_16.txt>nul
rodin3 scheme4 16>..\out\t4_16.txt>nul
rodin3 scheme5 16>..\out\t5_16.txt>nul
rodin3 scheme6 16>..\out\t6_16.txt>nul
rodin3 scheme7 16>..\out\t7_16.txt>nul
rodin3 scheme8 16>..\out\t8_16.txt>nul
rodin3 scheme9 16>..\out\t9_16.txt>nul
rodin3 scheme10 16>..\out\t10_16.txt>nul
rodin3 scheme11 16>..\out\t11_16.txt>nul
rodin3 scheme12 16>..\out\t12_16.txt>nul
rodin3 scheme13 16>..\out\t13_16.txt>nul
rodin3 scheme14 16>..\out\t14_16.txt>nul

rem comparaison of all SCHEMx files, done for 16 bits
Echo Comparing results (16 bits) with expected file: result into: "batchres.txt".
fc ..\out\schem*.* ..\out\etalon\*.* >..\out\etalon\batchres.txt

Echo.
Echo Working in Scheme*, in 32 bits
rodin3 scheme0 32>..\out\t0_32.txt>nul
rodin3 scheme1 32>..\out\t1_32.txt>nul
rodin3 scheme2 32>..\out\t2_32.txt>nul
rodin3 scheme3 32>..\out\t3_32.txt>nul
rodin3 scheme4 32>..\out\t4_32.txt>nul
rodin3 scheme5 32>..\out\t5_32.txt>nul
rodin3 scheme6 32>..\out\t6_32.txt>nul
rodin3 scheme7 32>..\out\t7_32.txt>nul
rodin3 scheme8 32>..\out\t8_32.txt>nul
rodin3 scheme9 32>..\out\t9_32.txt>nul
rodin3 scheme10 32>..\out\t10_32.txt>nul
rodin3 scheme11 32>..\out\t11_32.txt>nul
rodin3 scheme12 32>..\out\t12_32.txt>nul
rodin3 scheme13 32>..\out\t13_32.txt>nul
rodin3 scheme14 32>..\out\t14_32.txt>nul
```

```
rem comparaison of all SCHEMx files, done for 32 bits
Echo Comparing results (32 bits) with expected file: result into: "batchres.txt".
fc ..\out\schem* ..\out\etalon\* >>..\out\etalon\batchres.txt

Echo.
Echo Testing **Valid** Integer for 16 bits
rodin3 TestInteger16 16> ..\out\TestInteger16>nul

ECHO.>>..\out\etalon\batchres.txt
ECHO Testing **Valid** Integer in 16 bits>>..\out\etalon\batchres.txt
Echo Comparing with expected results for valid Integer16
fc ..\out\TestInteger16 ..\out\etalon\testInteger16 >>..\out\etalon\batchres.txt

Echo.
Echo Testing **Valid** Integer for 32 bits
rodin3 TestInteger32 32> ..\out\TestInteger32>nul
ECHO.>>..\out\etalon\batchres.txt
ECHO Testing **valid** Integer in 32 bits >>..\out\etalon\batchres.txt
Echo Comparing with expected results for valid Integer32
fc ..\out\TestInteger32 ..\out\etalon\testInteger32 >>..\out\etalon\batchres.txt

ECHO.
Echo Testing Number for 16 bits
rodin3 testNum 16> ..\out\testnum_16.txt>nul
ECHO.>>..\out\etalon\batchres.txt
ECHO Testing number in 16 bits>>..\out\etalon\batchres.txt
Echo Comparing with expected results for TestNum in 16 bits
fc ..\out\testnum.err ..\out\etalon\testnum16.err >>..\out\etalon\batchres.txt

Echo.
Echo Testing Number for 32 bits
rodin3 testNum 32> ..\out\testnum_32.txt>nul
ECHO.>>..\out\etalon\batchres.txt
ECHO Testing number in 32 bits>>..\out\etalon\batchres.txt
Echo Comparing with expected results for TestNum in 32 bits
fc ..\out\testnum.err ..\out\etalon\testnum32.err >>..\out\etalon\batchres.txt

rem Comparaison of the resul of the comparaisonS!
Echo.
Echo.
Echo Comparing result of all comparaison to expected result.
Echo File was recreated>..\out\etalon\result.txt
Echo.>>..\out\etalon\result.txt
Echo.>>..\out\etalon\result.txt

fc ..\out\etalon\batchres.txt ..\out\etalon\master.txt >>..\out\etalon\result.txt
Echo.

ECHO Press enter to see the result
Echo.>>..\out\etalon\result.txt
Echo Result of test dated:>>..\out\etalon\result.txt
date >>..\out\etalon\result.txt
echo.>>..\out\etalon\result.txt
echo.|time>>..\out\etalon\result.txt
Echo *** End of typed file ***>>..\out\etalon\result.txt

cls
Echo Result of execution is:
type ..\out\etalon\result.txt |more
pause

goto :fin
```

```
rem No Exec File
:NoRodin3
cls
Echo.
Echo The file Rodin3.exe cannot be found!
Echo.
Echo Please verify that you are in same directory as the rodin3.exe file
Echo or that the file Rodin3.exe exist.

:fin
cd..
cd nw

echo on>nul
echo off
r: >nul
cd\ >nul
cd rodin>nul
cd noweb >nul
cls

type r:\bat\affiche.txt

fin
```

ANNEXE

4 : Fichier de données XML

```
<mtTest>
   <!-- option de configuration de MTest -->
   <mtConfig>
      <!-- Limite de la difference du nombre de caracteres-->
      <maxDiffCharacter>100</maxDiffCharacter>
      <!-- Est-ce que les chemins(path) sont case sensitive? -->
      <pathCaseSensitive>YyNn</pathCaseSensitive>

      <!-- Est-ce que les fichiers output sont case sensitive? -->
      <fileCaseSensitive>YyNn</fileCaseSensitive>

      <!-- L'option "singleTokenForTextComp" definit les caracteres qui seront
           reconnus comme un jeton entier.
           Ceci permet de valider une comparaison de mode texte avec une equivalence
           de la multiplicite des blancs.

           EX1: en comparaison mode texte
           "Foo Bar" est equivalent est "Foo      bar"

           Ex2: FOO:BAR  equivalent a "FOO    :  BAR"
           Car les jetons sont ->FOO<- ->:<- ->BAR<-

           Note: le ->\<- ne doit pas etre double.
           =+-/*;:(){}[]\"'\~\<
         -->
      <singletonTokenForTextComp>
          =+-/*\;:(){}[]"'~&gt;&lt;!%?&amp;
      </singletonTokenForTextComp>

   </mtConfig>

   <!-- Liste des tools disponibles-->
   <!-- Peut etre multiple-->
   <mtTools>
      <tool name="nameOfTool_1">
         <!-- Extension du fichier qui doit etre traite par l'tool-->
         <toolOutType>bmp</toolOutType>

         <!-- Localisation of the tool. Local only -->
         <toolPath />

         <!-- Liste des parametres a utiliser: %file% indique le positionnement du fichier
         inclura l'extention. Ex: %file% = foo.bar
          IE: jpg name=%file% mem=32-->
         <toolParam>%file%</toolParam>

      </tool>
      <tool name="nameOfTool_2">

        <!-- Extension du fichier qui doit etre traite par le tool-->
        <toolOutType>bin</toolOutType>

        <!-- Localisation of the tool. Local only -->
        <toolPath />

        <!-- Liste des parametres a utiliser: %file% indique le positionnement du fichier
        inclura l'extention. Ex: %file% = foo.bar
        IE: jpg name=%file% mem=32-->
        <toolParam>%file%</toolParam>
```

```
   </tool>

</mtTools>
<!-- Contient la liste des noms des Systemes d'operation -->
<project name="nameProject_1">
   <!-- La description du project -->
   <descProject>Ici la description du project 1</descProject>
   <refNb>P1_refNb</refNb>
   <!-- Adresse courriel pour le project  -->
   <mailReportToProject>Foo@bar.com</mailReportToProject>
   <groupSoftware>
      <!-- Decription du Group de software en action concurrente -->
      <descGroupSoftware> La description du Group</descGroupSoftware>
      <!-- Adresse courriel pour le Group de software -->
      <mailReportToGroup>Foo@bar.com</mailReportToGroup>
      <!-- Action a effectuer avant l'activation du Group.
           Il peut y avoir plusieurs actions
      -->
      <!-- L'ordre d'activation est similaire a l'ordre d'apparition-->
      <preActionTestGroup> action to be done first</preActionTestGroup>
      <!-- Action a effectuer apres la terminaison du Group de software.
           Il peut y avoir plusieurs actions
      -->
      <!-- L'ordre d'activation est similaire a l'ordre d'apparition-->
      <postActionTestGroup> action to be done afterTestGroup</postActionTestGroup>
      <software name="namesoftwareInGroup_1">
         <!-- Description du software -->
         <descSoftware>Ceci est la description d&apos;un software_1</descSoftware>
         <!-- Courriel du responsable du software -->
         <mailReportToSoftware>
            AdrCourriel si bug dans _ce_ software_1
         </mailReportToSoftware>
         <!--Chemin de l'executable -->
         <pathBin>./bin/logicie_l.exe</pathBin>
         <!-- -->
         <pathUSEzipFrom/>
         <!-- Numero de reference du software -->
         <refNb/>
         <!-- Fichier generique de sortie.  Ex: gestion de memoire-->
         <Generique>
            <!-- Localisation  du fichier generique de sortie-->
            <generiqueSortie/>
            <!-- Localisation du fichier etaton du fichier generique de sortie-->
            <generiqueEtalon/>
         </Generique>
         <test nameTest="nameTest_1">
            <!-- Description du test_1-->
            <descTest/>
            <!--Localisation du fichier d'entree, peut etre multiple-->
            <!-- L'ordre d'apparition indique l'ordre de substitution dans les
                 parametres.
            -->
            <in>.</in>
            <in>.</in>
            <!-- Localisation du fichier de sortie, peut etre multiple-->
            <!-- L'ordre d'apparition indique l'ordre de substitution dans les
                 parametres.
            -->
            <out>.</out>
            <out>.</out>
            <param>Foo %in% %in% %out% %out%</param>
            <etalon>.</etalon>
         </test>
      </software>
   </groupSoftware>
   <software name="namesoftware_1">
```

```
<!-- Description du software -->
<descSoftware>Ceci est la description d&apos;un software_1</descSoftware>
<!-- Courriel du responsable du software -->
   <mailReportToSoftware>AdrCourriel si bug dans _ce_
    software_1</mailReportToSoftware>
<!-- Logiciel Java: Boolean yYnN -->
<bJava>N</bJava>

<!-- Le nom du systeme d'operation pour le logiciel-->
<OSName></OSName>
<!--Chemin de l'executable
Le repertoire de travail (working directory) est le repertoire
contenant le binaire a executer.-->
<pathBin>../../SoftwareTest/Fonctions/bin/Fonctions.exe</pathBin>

<workingDirectory></workingDirectory>

<!-- Chemin du repertoire qui contiendra tous les rapports
     des tests de regressions Si le chemin specifie n'esistepas, il est cree.
  -->
<pathReport>../../SoftwareTest/Fonctions/Test/Rapport/</pathReport>

<!--Chemin+Fichier contenant le build a extraire avant de faire les tests.
peut etre vide/absent. Est un Singleton-->

<buildPathFrom></buildPathFrom>

<!--buildPathTo
Repertoire cible pour l'extraction du fichier "Build"-->

<buildPathTo></buildPathTo>

<!--Variable d'environnement
format "Foo=FOOBAR"
si plusieurs variables d'environnement, faire plusieurs tags. -->
<envVariables>
   <envVariable></envVariable>
</envVariables>

<!-- -->

<pathUseFrom>./use</pathUseFrom>

<pathUseTo>./bin</pathUseTo>

<!-- Numero de reference du software -->

<refNb>refNumber 10101</refNb>

<!-- Fichier generique de sortie.  Ex: gestion de memoire-->
<Generique>
   <!-- Localisation  du fichier generique de sortie-->
   <generiqueSortie/>
   <!-- Localisation du fichier etaton du fichier generique de sortie-->
   <generiqueEtalon/>
</Generique>
<test name="nameTest_1">
   <!-- Description du test_1-->
   <descTest>Description du test 1</descTest>

   <!-- Localisation du fichier d'entree-->
   <!-- Doit etre present -->
   <in>../../SoftwareTest/Fonctions/Test/in/file_1.txt</in>
   <!-- Localisation du fichier de sortie, peut etre multiple-->
   <!-- Doit etre present -->
   <out>../../SoftwareTest/Fonctions/Test/out/file_1.out</out>
   <param>%in% %out%</param>
```

```
            <etalon>../../SoftwareTest/Fonctions/Test/etalon/file_1.out</etalon>
         </test>
      </software>
   </project>
</mtTest>
```