UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE
APPLIQUÉES

PAR
PIERRE-LUC VINCENT

# TESTS DE RÉGRESSION DANS LES SYSTÈMES ORIENTÉS OBJET : UNE APPROCHE BASÉE SUR LES MODÈLES

Novembre 2009

Université du Québec à Trois-Rivières

Service de la bibliothèque

Avertissement

L'auteur de ce mémoire ou de cette thèse a autorisé l'Université du Québec à Trois-Rivières à diffuser, à des fins non lucratives, une copie de son mémoire ou de sa thèse.

Cette diffusion n'entraîne pas une renonciation de la part de l'auteur à ses droits de propriété intellectuelle, incluant le droit d'auteur, sur ce mémoire ou cette thèse. Notamment, la reproduction ou la publication de la totalité ou d'une partie importante de ce mémoire ou de cette thèse requiert son autorisation.

**TESTS DE RÉGRESSION DANS LES SYSTÈMES ORIENTÉS OBJET : UNE APPROCHE BASÉE SUR LES MODÈLES**

Pierre-Luc Vincent

## SOMMAIRE

L'évolutivité des systèmes orientés objet est devenu un enjeu économique important. Plusieurs de ces systèmes ont, en effet, nécessité un travail de longue haleine. Il devient alors indispensable de leur assurer une durée de vie conséquente et surtout de faciliter leur évolution. L'évolution de ces systèmes est souvent inéluctable et reste une tâche très complexe. Les systèmes orientés objet sont caractérisés par les dépendances qui existent entre leurs classes. La complexité et la diversité de ces dépendances rendent l'évolution de ces systèmes difficile. Leur gestion durant le processus de développement et de maintenance est une tâche très complexe mais néanmoins indispensable. Il est en effet très difficile d'évaluer intuitivement l'impact d'une modification quelconque d'un composant sur le reste de l'application et d'identifier la ou les parties du système qui doivent être re-testées. Les dépendances entre les classes d'un système ont une incidence directe sur ce problème. L'approche proposée dans ce mémoire permet de supporter les tests de régression dans les systèmes orientés objet en se basant sur les cas d'utilisation et plus précisément sur les modèles UML qui les décrivent (diagrammes d'état de cas d'utilisation et diagrammes d'interaction). Il s'agit d'une approche complètement indépendante des langages de programmation. L'approche permet d'identifier les cas d'utilisations ayant subis une modification et par conséquent les parties du système qui doivent être re-testées. Un outil a été développé afin de supporter l'approche. L'approche et l'outil sont illustrés à l'aide d'une étude de cas.

**REGRESSION TEST SELECTION IN OBJECT-ORIENTED SYSTEMS:**

**A MODEL BASED APPROACH**

Pierre-Luc Vincent

## ABSTRACT

The evolution of object-oriented systems has become an important economic stake. Several of these systems required a huge amount of work. It is then important to facilitate their evolution, which is still a very complex task. Object-oriented systems are characterized by the dependencies existing between their classes. The complexity of these dependencies makes this task very difficult. It is, in fact, very difficult to assess intuitively the impact of modifying a given component on the rest of the system and to identify the parts of the system that must be retested. The dependencies between classes have a direct impact on this problem. The proposed approach supports regression test selection in object-oriented systems based on use cases models, and more precisely the UML models that describe them (state and interaction diagrams). The approach is entirely independent of programming languages. The approach allows identifying use cases that have been modified and consequently the parts of the system that must be retested. A tool has been developed in order to automate the process. The approach and its supporting tool are illustrated using a case study.

# REMERCIEMENTS

J'aimerais tout d'abord remercier toute ma famille de m'avoir supporté tout au long de mes études.

J'aimerais aussi remercier mes directeurs de recherche Linda Badri et Mourad Badri pour leur soutient, leur grande disponibilité et leur sens de l'organisation. Merci beaucoup.

J'ai une pensée spéciale pour mon père qui nous a quittés il y a bientôt trois ans. Merci de m'avoir transmit de si belles valeurs que sont l'honneur, la détermination et la persévérance.

# TABLE DES MATIÈRES

Page

SOMMAIRE ..... ii
ABSTRACT ..... iii
REMERCIEMENTS ..... iv
TABLE DES MATIÈRES ..... v
LISTE DES TABLEAUX ..... viii
LISTE DES FIGURES ..... ix
INTRODUCTION ..... 1
Évolution du logiciel ..... 1
Problématique ..... 2
Approche ..... 2
Organisation ..... 3
CHAPITRE 1 ÉTAT DE L'ART ..... 4
1.1 Les approches de l'ère procédurale ..... 4
1.2 Les approches de l'ère objet ..... 8
1.3 Les approches basées sur les modèles ..... 12
1.4 Conclusion ..... 14
CHAPITRE 2 CRITÈRES D'ÉVALUATION ..... 15
2.1 État de l'art ..... 15
2.2 Observations ..... 17
2.3 Description de la méthode d'évaluation choisi ..... 19
2.4 Conclusion ..... 20
CHAPITRE 3 COMPARAISON DE PROGRAMMES ..... 21
3.1 Approche ..... 21
3.2 Analyse lexicale et syntaxique ..... 21
3.3 Arbre syntaxique abstrait ..... 22
3.4 Processus de comparaison ..... 23
3.5 Conclusion ..... 23
CHAPITRE 4 MODÈLES UML ..... 24
4.1 Cas d'utilisation ..... 24
4.2 Diagramme d'états de cas d'utilisation ..... 26

4.3 Diagramme de collaboration ........ 27
4.4 Liens entre le diagramme d'états de cas d'utilisation et le diagramme de collaboration ... 27
4.5 Conclusion ........ 28
CHAPITRE 5 MÉTHODOLOGIE DE L'APPROCHE ........ 29
5.1 Méthodologie de l'approche ........ 29
5.2 Conclusion ........ 34
CHAPITRE 6 GÉNÉRATION DES SÉQUENCES DE TESTS ........ 35
6.1 Introduction ........ 35
6.2 Concepts théoriques ........ 35
6.2.1 Graphe de contrôle réduit aux appels ........ 35
6.2.2 Arbre des messages ........ 37
6.2.3 Séquence principale ........ 38
6.3 Processus de génération des séquences de tests ........ 39
6.4 Exemple théorique ........ 41
6.5 Génération automatique des tests ........ 45
6.5.1 Objectif ........ 45
6.5.2 Méthode ........ 45
6.6 Conclusion ........ 47
CHAPITRE 7 EXPÉRIMENTATION ........ 48
7.1 Protocole d'expérimentation ........ 48
7.1.1 Objectif ........ 48
7.1.2 Démarche ........ 48
7.1.3 Critères ........ 49
7.1.4 Protocole ........ 50
7.2 Étude de cas et résultats ........ 51
7.2.1 Traiter une vente ........ 52
7.2.2 Sécurité ........ 54
7.2.3 Ajout, modification et suppression d'un utilisateur ........ 55
7.2.4 Analyser les ventes ........ 55
7.2.5 Démarrer le système ........ 56
7.2.6 Arrêter le système ........ 57
7.2.7 Gérer les tables ........ 57
7.2.8 Version V2 ........ 58
7.2.8.1 Résultats de l'itération 1 (Passage de la version V1 à V2) ........ 60
7.2.9 Version V3 ........ 60
7.2.9.1 Résultats de l'itération 2 (Passage de la version V2 à V3) ........ 61
7.2.10 Version V4 ........ 62
7.2.10.1 Résultats de l'itération 3 (Passage de la version V3 à V4) ........ 64
7.2.11 Version V5 ........ 64
7.2.11.1 Résultats de l'itération 4 (Passage de la version V4 à V5) ........ 65

7.2.12 Illustration des résultats ... 66
7.3 Conclusion ... 67
CHAPITRE 8 PRÉSENTATION DE L'OUTIL ... 68
8.1 Module d'analyse statique du code ... 69
8.2 Module d'analyse de modèles UML ... 70
8.3 Fonctionnement de l'application ... 74
8.4 Conclusion ... 76
CONCLUSIONS ... 77
BIBLIOGRAPHIE ... 79

# LISTE DES TABLEAUX

Page

Table I Analyse des diverses approches de tests de régression au cours de l'ère procédurale....... 6
Table II Efficacité de la détection des fautes selon le niveau de granularité................................. 8
Table III Condensé des différentes techniques d'évaluation décrites dans l'état de l'art. ........... 18
Table IV Grammaire XML pour les diagrammes. ........................................................................ 70
Table V Grammaire XML pour les diagrammes de collaboration. ............................................. 71

# LISTE DES FIGURES

Page

Figure 1 Exemple de graphe de relations objet ... 9
Figure 2 Exemple de génération du graphe acyclique ... 12
Figure 3 Exemple d'arbre syntaxique abstrait [CMSC01]. ... 22
Figure 4 Cas d'utilisation 'Traiter une vente' [Larman01]. ... 25
Figure 5 Diagramme d'états du cas d'utilisation 'traiter une vente' [Larman01]. ... 26
Figure 6 Diagramme de collaboration pour le scénario 'saisirArticle' [Larman01]. ... 28
Figure 7 Exemple théorique de la méthodologie de l'approche ... 31
Figure 8 Modèle général de la méthodologie proposée ... 33
Figure 9 Exemple de graphe de contrôle réduit aux appels ... 37
Figure 10 Exemple de transition allant d'un pseudo code jusqu'à l'arbre des messages ... 39
Figure 11 Diagramme de collaboration pour la méthode 'saisirArticle'. ... 41
Figure 12 Génération de la séquence principale pour la collaboration 'saisirArticle'. ... 42
Figure 13 Diagramme d'états du cas d'utilisation 'traiter une vente'. ... 43
Figure 14 Génération de la séquence principale pour le cas d'utilisation. ... 43
Figure 15 Séquence finale de tests pour l'exemple. ... 44
Figure 16 Exemple de génération automatique des tests ... 46
Figure 17 Diagramme d'état du cas d'utilisation 'Traiter une vente'. ... 52
Figure 18 Diagramme de collaboration du message 'saisirArticle'. ... 53
Figure 19 Diagramme de collaboration du message 'créerPaiement'. ... 53
Figure 20 Diagramme d'état du cas d'utilisation 'Sécurité' ... 54
Figure 21 Diagramme de collaboration du message 'vérifierAuthentification' ... 54
Figure 22 Diagrammes d'états des cas d'utilisation 'Ajouter un utilisateur', 'Modifier un utilisateur' et 'Supprimer un utilisateur', de haut en bas respectivement ... 55
Figure 23 Diagramme d'états du cas d'utilisation 'Analyser les ventes'. ... 55
Figure 24 Diagramme de collaborations du message 'AnalyserVentes' ... 56
Figure 25 Diagramme d'états du cas d'utilisation 'DémarrerSystème'. ... 56
Figure 26 Diagramme d'états du cas d'utilisation 'Arrêter système' ... 57
Figure 27 Diagramme d'états du cas d'utilisation 'Gérer les tables'. ... 57
Figure 28 Diagramme d'états modifié du cas d'utilisation 'Traiter une vente'. ... 58
Figure 29 Diagramme de collaborations (nouveau) du message 'créerPaiementACrédit'. ... 58
Figure 30 Diagramme de collaboration modifié de 'créerPaiementÀCrédit' ... 61
Figure 31 Diagramme de collaboration de 'refuserAccès'. ... 61
Figure 32 Diagramme d'états modifié du cas d'utilisation 'traiter une vente'. ... 63

Figure 33 Diagramme de collaboration modifié de la méthode 'saisirUnArticle', auparavant nommée 'saisirArticle'. .... 63
Figure 34 Comparaison du nombre de séquences sélectionnées entre l'approche proposée et deux autres approches. .... 66
Figure 35 Taux de réutilisation des cas de test selon l'approche proposée lors des expérimentations. .... 67
Figure 36 Architecture générale de l'outil .... 68
Figure 37 Fonctionnement général du module d'analyse statique. .... 69
Figure 38 Capture d'écran de l'utilitaire qui permet de saisir les éléments d'un diagramme d'état de cas d'utilisation et de générer automatiquement le fichier XML correspondant. .... 72
Figure 39 Capture d'écran de l'utilitaire qui permet de saisir les éléments d'un diagramme de collaboration et de générer automatiquement le fichier XML correspondant. .... 73
Figure 40 Exemple concret de fichier XML représentant un diagramme d'états de cas d'utilisation (L'exemple correspond aux données saisies tel qu'illustré à la figure 36). .... 73
Figure 41 Exemple concret de fichier XML représentant un diagramme de collaboration (L'exemple correspond aux données saisies tel qu'illustré à la figure 37). .... 74
Figure 42 Interface utilisateur de l'application. .... 75
Figure 43 Exemple de résultat. .... 76

# INTRODUCTION

Au cours des deux dernières décennies, le paradigme objet a connu une grande période d'effervescence. En effet, un grand nombre des applications logicielles que nous utilisons dans nos vies au quotidien sont des systèmes à objets. L'approche objet a, en effet, introduit plusieurs nouveaux concepts permettant le développement d'applications plus larges, plus robustes et de façon plus efficace. Dans un système à objets, l'unité est représentée par la classe. Grâce au principe de l'encapsulation, les données et le comportement d'un concept font partie intégrante de la classe et sont protégés au moyen d'un mécanisme de visibilité. Afin de réaliser les besoins fonctionnels du logiciel, les classes entretiennent diverses dépendances et collaborent les unes avec les autres. Elles se partagent des responsabilités. Les classes agissent, selon les responsabilités qui leur ont étés assignées, de façon similaire à un fournisseur de services. Les relations de dépendance entre classes soulèvent cependant une problématique : si une classe A dépend d'une classe B et que cette dernière est modifiée suite à une activité de maintenance par exemple, alors il est possible que le fonctionnement de la classe A soit indirectement affecté.

## Évolution du logiciel

L'évolution de certains systèmes logiciels est souvent inéluctable et reste une tâche très complexe. En effet, les études de Lehman et al. [Lehman 80] tendent à démontrer, entre autres, les deux phénomènes suivants :

i. Le changement continu : un système logiciel qui est utilisé doit être continuellement adapté ou il deviendra progressivement moins satisfaisant;
ii. L'augmentation de la complexité : un système logiciel qui est modifié a tendance à devenir de moins en moins structuré et de plus en plus complexe.

Il devient donc important de supporter l'évolution de ces systèmes tout en réduisant leur coût et leur délai, en simplifiant leur maintenance et en facilitant leur réutilisation.

## Problématique

La complexité et la diversité des dépendances pouvant exister entre les différents composants d'un système logiciel rendent cette tâche difficile. Les dépendances entre classes dans les systèmes orientés objet (en particulier en termes de contrôle et de données) constituent une de leurs principales caractéristiques. Ces dépendances ont un effet important sur leur qualité, en particulier sur leur test. La gestion de ces dépendances durant le processus de développement et de maintenance est une tâche très complexe mais néanmoins indispensable. Il est très difficile, voire même impossible dans certains cas, d'évaluer intuitivement et a priori l'impact d'une modification quelconque d'un composant, par exemple, sur le reste de l'application et d'identifier la ou les parties du système qui doivent être re-testées. Les dépendances entre les classes d'un système ont une incidence directe sur ce problème.

## Approche

Il s'agit, dans ce projet, d'implémenter une nouvelle approche pour réaliser les tests de régression dans les systèmes orientés objet et de l'expérimenter. L'approche en question est basée sur les cas d'utilisation et plus précisément sur les modèles UML qui les décrivent (diagrammes d'états de cas d'utilisation et diagrammes d'interaction). L'approche permet d'identifier les cas d'utilisations ayant subit une modification et d'identifier les parties d'un système qui doivent être re-testées, la sélection des cas de test appropriés (dans une batterie complète existante) ainsi que la détermination des nouveaux cas de test (extension de la batterie existante). Un outil a été réalisé afin que l'approche soit entièrement automatisée. L'approche proposée et l'outil la supportant ont été évalués, sur une étude de cas concrète, en fonction de critères bien définis suite à une revue de la littérature spécialisée.

**Organisation**

Ce mémoire se divise en onze chapitres. Le chapitre 1 résume l'état de l'art des travaux portant sur le test de régression. Le chapitre 2 présente une revue de la littérature en ce qui concerne les critères d'évaluation des approches supportant le test de régression. Une synthèse est par la suite présentée et les critères qui seront utilisés dans le cadre de ce travail sont définis. Le chapitre 3 présente la méthodologie de l'approche proposée. Le chapitre 4 présente les concepts relatifs à l'analyse de programmes utilisés dans le cadre de ce travail. Le chapitre 5 décrit les diagrammes UML utilisés ainsi que les éléments qui les composent. Le chapitre 6 présente le processus de génération des séquences de test. Des concepts théoriques y sont présentés ainsi que l'algorithme de base de l'approche et un exemple concret. Le chapitre 7 présente le processus général de génération de la batterie de tests de régression à partir des séquences de tests de base. Le chapitre 8 présente le protocole d'expérimentation. Le chapitre 9 décrit l'étude de cas utilisée lors des expérimentations ainsi que les quatre versions subséquentes où divers changements sont apportés afin d'expérimenter l'approche. Il présente également les résultats de ces expérimentations ainsi que des graphiques les résumant. Le chapitre 10 présente l'outil qui a été développé afin d'automatiser l'approche. Le fonctionnement des différents modules y est présenté. De plus, la description formelle des modèles UML en format XML est décrite à l'aide de grammaires et des exemples concrets sont aussi présentés. Finalement, une conclusion générale est présentée.

# CHAPITRE 1

# ÉTAT DE L'ART

Plusieurs approches portant sur les tests de régression ont été développées. Ces diverses approches peuvent être catégorisées en trois grandes familles : les approches de l'ère procédurale, les approches de l'ère objet et les approches basées sur les modèles. Les premiers balbutiements au niveau des tests de régression ont porté sur des programmes de type procéduraux. Plusieurs de ces approches ont été adaptées par la suite pour être applicables aux systèmes orientés objet. Ensuite, plusieurs approches dédiées aux systèmes orientés objet ont fait leur apparition. Au même moment, plusieurs avancées significatives ont été réalisées concernant l'analyse et la modélisation orientée objet. Les nouveau processus de développement sont agiles, légers et adaptatifs et mettent en lumière l'importance des premières phases du cycle de développement et des modèles sur lesquelles elles se basent. Les besoins fonctionnels et le comportement souhaité du logiciel sont de plus en plus représentés par la notation UML. Les modèles sont devenus des éléments clés dans les processus d'ingénierie logicielle. À ce jour, il n'y a que très peu d'approches portant sur les tests de régression qui sont basées sur les modèles UML.

## 1.1 Les approches de l'ère procédurale

Depuis le début des années 1990, plusieurs travaux portant sur les tests de régression ont été publiés. La plupart des travaux qui ont été produits au début de la décennie 1990 s'appliquent au paradigme procédural. Gupta et al. [Gupta92] proposent l'utilisation du découpage de programme (slicing) afin d'identifier les relations d'utilisation et ainsi de générer les tests à exécuter sans avoir à maintenir une suite de test. Agrawal et al. [Agrawal93] utilisent l'instrumentation et l'analyse dynamique de programmes afin de produire un découpage de programme lors de l'exécution et ainsi choisir les cas de test à exécuter. Bates et al. [Bates93] utilisent les graphes de dépendance afin de modéliser les dépendances au niveau du contrôle et des données d'un programme.

Le découpage de programme (Slicing) est ensuite utilisé sur les graphes afin de sélectionner les cas de test à exécuter. Binkley et al. [Binkley95] utilisent les graphes de dépendance système et le découpage de programme inter-procédural afin de sélectionner les cas de test. Chen et al. [Chen94] présentent l'outil TestTube, basé sur une approche qui combine l'analyse statique et dynamique de programmes afin de sélectionner les cas de test de régression de systèmes écrits en langage C. Les fonctions, les types, les variables et les macros qui sont couverts par chaque cas de test sont identifiés et insérés dans une base de données. Le code est ensuite instrumenté et exécuté. Rothermel et al. [Rothermel94, Rothermel96] présentent une plateforme qui permet d'évaluer les diverses approches existantes. La table I ([Rothermel96]) résume bien les différentes approches utilisées lors de l'ère procédurale. Dans [Rothermel97-1], Rothermel et al. présentent une technique qui utilise à la fois l'analyse statique et dynamique de programmes. Le code est instrumenté pour que, lors de l'exécution de chaque cas de test, le chemin du graphe de contrôle qui correspond au chemin traversé soit enregistré. Ensuite, les graphes de contrôle sont parcourus et les tests qui exécutent du code modifié sont sélectionnés. Ce travail a servi de base à plusieurs autres travaux. Rothermel et al. proposent dans [Rothermel97-2] un outil nommé DejaVu afin d'implémenter et d'automatiser l'approche précédente. Ball et al. [Ball98] proposent quelques améliorations afin que l'algorithme de Rothermel et al. [Rothermel97-1] soit plus précis et efficace. Forgács et al. [Forgács98] proposent une méthode qui se base sur celle de Agrawal et al. [Agrawal93].

L'instrumentation et le découpage de programmes permettent de générer un graphe uni qui se compose du graphe de dépendances statique et du graphe de dépendances dynamique. Korel et al. [Korel98] proposent une méthode qui génère des données de test, exécute les cas de test sur les deux versions et sélectionne seulement les cas de test pour lesquels les résultats sont différents. Binkley et al. [Binkley99] établissent la liste des différentes approches qui se basent sur le découpage de programmes. La conclusion est qu'il y a un manque de cohésion entre ces méthodes puisque pour un même programme, chaque approche donne un résultat différent. Graves et al. [Graves98] présentent une étude empirique qui permet de comparer diverses approches.

| TECHNIQUE | INCLUSIVENESS | PRECISION | EFFICIENCY | GENERALITY |
| --- | --- | --- | --- | --- |
| LINEAR EQUATION (minimization) (intra) | unsafe (all categories) | selects non-mt tests | worst case: exponential in $\|P\|$<br>in practice: unknown<br>correspondence: $O(max(\|P\|,\|P'\|)^2)$ | level: intra<br>mods: not control flow<br>criteria: control/dataflow<br>requires: segment traces, linear equation solver |
| LINEAR EQUATION (non-minimization) (inter) | safe for controlled regression testing | selects non-mt tests<br>selects all tests through modified procedures | worst case: exponential in $\|P\|$<br>in practice: unknown<br>correspondence: $O(max(\|P\|,\|P'\|)^2 * log(max(\|P\|,\|P'\|)))$ | level: inter<br>mods: handles all<br>criteria: control/dataflow<br>requires: function traces, linear equation solver |
| SYMBOLIC EXECUTION | not safe (for deletions) | selects non-mt tests | worst case: exponential in $\|P\|$ (may not terminate)<br>in practice: unknown<br>correspondence: $O(max(\|P\|,\|P'\|)^2 * log(max(\|P\|,\|P'\|)))$ | level: intra/inter<br>mods: not deletions<br>criteria: partition<br>requires: symbolic execution |
| PATH ANALYSIS | not safe (for deletions or additions) | selects no non-mt tests | worst case: exponential in $\|P\|$<br>in practice: unknown<br>correspondence: not required | level: intra<br>mods: not deletions/additions<br>criteria: path<br>requires: statement traces, algebraic design |
| DATAFLOW (incremental) | not safe (all categories) | selects non-mt tests | worst case: $O(\|T\| * \|P'\|^2)$ per modification<br>in practice: unknown<br>correspondence: not required | level: intra/inter<br>mods: only dataflow affecting<br>criteria: dataflow<br>requires: basic block traces, static and incremental dataflow analysis tools |
| PROGRAM DEPENDENCE GRAPH | not safe (for deletions) | selects non-mt tests<br>less precise than SDG technique | worst case: $O(\|T\| * (max(\|P\|,\|P'\|))^3)$ or $O(\|T\| * (max(\|P\|,\|P'\|))^4)$<br>in practice: unknown<br>correspondence: $O(max(\|P\|,\|P'\|)^2)$ | level: intra<br>mods: not deletions<br>criteria: PDG<br>requires: statement traces, control dependence, slicing, dataflow analysis tools |
| SYSTEM DEPENDENCE GRAPH | not safe (for deletions) | selects non-mt tests<br>more precise than PDG technique | worst case: $O(\|T\| * (max(\|P\|,\|P'\|))^3)$ or $O(\|T\| * (max(\|P\|,\|P'\|))^4)$<br>in practice: unknown<br>correspondence: $O(max(\|P\|,\|P'\|)^2)$ | level: intra/inter<br>mods: not deletions<br>criteria: PDG<br>requires: statement traces, control dependence, slicing, dataflow analysis tools |

| TECHNIQUE | INCLUSIVENESS | PRECISION | EFFICIENCY | GENERALITY |
|---|---|---|---|---|
| MODIFICA-<br>TION<br>BASED | not safe<br>(all categories)<br>(minimization) | unknown | worst case: $O(\lvert T\rvert \times \lvert P'\rvert^3)$ per<br>logical modification for<br>analysis; test selection<br>is not automated.<br>in practice: unknown<br>correspondence: $O(max(\lvert P\rvert, \lvert P'\rvert)^2)$ | level: intra/inter<br>mods: doesn't handle all<br>criteria: none<br>requires: statement traces,<br>static/dynamic control and<br>data dependence analysis |
| FIREWALL | safe for controlled<br>regression<br>testing if T<br>is reliable | selects non-mt tests<br>selects all tests<br>through modified<br>procedures | worst case: $O(\lvert T\rvert \times (max(\lvert P\rvert, \lvert P'\rvert)))$<br>in practice: "efficient"<br>correspondence: $O(max(\lvert P\rvert, \lvert P'\rvert)^2$<br>$\times log(max(\lvert P\rvert, \lvert P'\rvert)))$ | level: inter<br>mods: handles all<br>criteria: none<br>requires: function traces |
| CLUSTER<br>IDENTIFI-<br>CATION | safe for p.r.t. | selects non-mt tests<br>less precise than graph<br>walk technique | worst case: $O(\lvert T\rvert \times max(\lvert P\rvert, \lvert P'\rvert)^3)$<br>in practice: unknown<br>correspondence: not required | level: intra<br>mods: handles all<br>criteria: none<br>requires: statement traces<br>CFGs, control dependence |
| SLICING | not safe<br>(some<br>categories) | selects non-mt tests | worst case: $O(\lvert T\rvert \times \lvert P'\rvert^3)$<br>per modification<br>in practice: unknown<br>correspondence: not required | level: intra<br>mods: doesn't handle all<br>criteria: none<br>requires: statement traces,<br>static/dynamic control and<br>data dependence analysis |
| GRAPH<br>WALK | safe for controlled<br>regression<br>testing | selects non-mt tests<br>(but not in practice)<br>most precise safe<br>technique | worst case: $O(\lvert T\rvert \times (max(\lvert P\rvert, \lvert P'\rvert))^2)$<br>in practice: $O(\lvert T\rvert \times min(\lvert P\rvert, \lvert P'\rvert))$<br>(without dataflow information)<br>correspondence: not required | level: intra/inter<br>mods: handles all<br>criteria: none<br>requires: statement traces<br>dataflow analysis (optional) |
| MODIFIED<br>ENTITY | safe for controlled<br>regression<br>testing | selects non-mt tests<br>selects all tests<br>through modified<br>procedures<br>less precise than graph<br>walk, cluster ident. | worst case: $O(\lvert T\rvert \times \lvert P\rvert)$<br>in practice: "efficient"<br>correspondence: $O(max(\lvert P\rvert, \lvert P'\rvert)$<br>$\times log(max(\lvert P\rvert, \lvert P'\rvert)))$ | level: inter<br>mods: handles all<br>criteria: none<br>requires: function traces,<br>code database |

Table I Analyse des diverses approches de tests de régression au cours de l'ère procédurale.
Source : Rothermel et al. [Rothermel96].

Kim et al. [Kim00] tentent de démontrer que la fréquence à laquelle sont effectués les tests de régression a un impact sur l'efficacité et la précision des différentes approches. Bible et al. [Bible01] présentent une étude empirique qui compare deux approches, une à granularité épaisse (TestTube) et l'autre à granularité fine (DejaVu). Les trois critères évalués sont la précision, l'efficacité et le temps d'analyse. Les résultats révèlent que DejaVu est plus précis dans la plupart des cas mais que le temps d'analyse est souvent plus long que de tout

retester. Rothermel et al. [Rothermel02] présente une étude empirique qui a pour objectif d'établir une corrélation entre la granularité de la suite de test et le rapport coût-bénéfice des tests de régression. La table II illustre l'efficacité de la détection des fautes selon le niveau de granularité. Dans ce contexte, le niveau de granularité indique dans quelle proportion les cas de tests sont filtrés. Un niveau de granularité fin signifie que très peu de cas sont sélectionnés. L'étude démontre que plus le niveau de granularité est fin, plus les résultats seront précis. Finalement, l'étude conclue que des compromis doivent être trouvés entre le niveau de granularité et le rapport coût bénéfice.

| | *Undetected Faults* | | *Avg. Faults Detected per Test* | |
|---|---|---|---|---|
| | emp-server | bash | emp-server | bash |
| G1 | 27 | 7 | 1.09 | 2.02 |
| G4 | 12 | 3 | 1.16 | 2.08 |
| G16 | 3 | 0 | 1.45 | 2.80 |
| G64 | 0 | 0 | 2.89 | 4.52 |

Table II Efficacité de la détection des fautes selon le niveau de granularité.

Source : Rothermel et al. [Rothermel02]

## 1.2 Les approches de l'ère objet

La décennie des années 1990 a aussi été une très grande période d'effervescence pour le paradigme orienté objet. Plusieurs approches pour les tests de régression appliqués aux programmes orientés objet ont été développées. Plusieurs de ces approches (la plupart) se basent sur une ou plusieurs approches énoncées précédemment. Kung et al. [Kung93] présentent un algorithme basé sur la technique du pare-feu pour les programmes orientés objet. La technique du pare-feu consiste à regrouper les modules qui doivent être re-testés suite à une modification. L'analyse statique du code est utilisée afin d'identifier, pour chaque version, les classes ayant subies des modifications. Afin d'identifier les classes qui sont susceptibles d'être affectées, on identifie les relations entre les classes comme suit :

a. l'héritage, décrit par la lettre I;
b. la composition, décrite par l'abréviation Ag;

c. les associations, décrites par l'abréviation As.

Par la suite, ces relations entre les classes sont représentées grâce à la construction d'un graphe dirigé. La figure 1 illustre un exemple de graphe de relations objet selon cette approche.

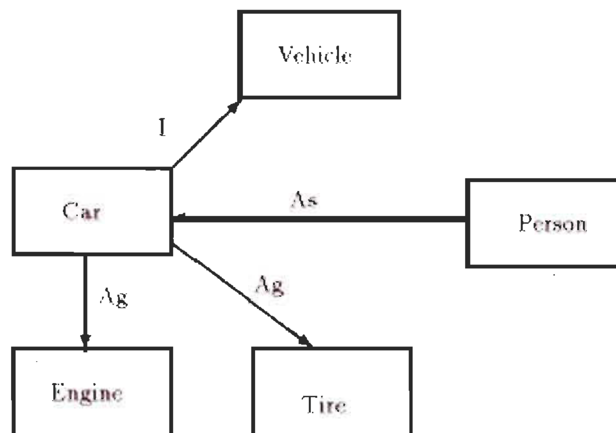


Figure 1 Exemple de graphe de relations objet

Source : Kung et al. [Kung93]

Il y a trois cas possibles où une classe peut être considérée comme étant affectée :

a. si A hérite de B et que B est modifiée A est affectée. (Héritage)
b. si B est composée de A et que B est modifiée, A est affectée. (Composition)
c. si A accède aux données membres de B ou que A appelle une méthode de B et que B est modifiée alors A est affectée. (Association)

Suite à l'analyse, toutes les classes marquées comme étant modifiées ou affectées doivent être re-testées. Abdullah et al. [Abdullah98] élaborent le concept de pare-feu présenté précédemment. La principale nouveauté est qu'une distinction est établie entre les changements de spécification et les changements au niveau du code. De plus, cette nouvelle

approche tient compte du polymorphisme et de la liaison dynamique. Une classe subit un changement de spécification si une ou plusieurs des modifications suivantes se produisent :

a. modifier une donnée membre (type, portée, initialisation);
b. modifier la signature d'une méthode (valeur de retour, paramètres);
c. ajout, suppression d'une méthode;
d. l'ajout ou le retrait d'un appel d'une méthode à une autre;
e. modifier la hiérarchie de la classe (héritage).

Une classe subit un changement au niveau code si une ou plusieurs des modifications suivantes se produisent :

a. changer le nom, l'interprétation, le domaine d'application d'une donnée membre;
b. modifier le code interne d'une méthode (incluant les variables locales à la méthode).

La représentation utilisée est le diagramme de relations des classes qui contient les classes, leur structure et les relations statiques entre les classes. Les relations suivantes sont considérées :

a. utilisation : B utilise A; A fait partie de la signature d'une méthode de B;
b. héritage : B hérite de la classe A;
c. composition : B est composée de la classe A;
d. association : B pointe sur la classe A.

La notion d'utilisation est nouvelle par rapport à l'approche de Kung et al. [Kung93]. White et al. [White05] présentent une extension à l'approche d'Abdullah et al. [Abdullah98]. Le pare-feu étendu a comme spécificité qu'il prend en compte, en plus des éléments du pare-feu standard : les variables globales, les cycles et les chemins. Skoglund et al. [Skoglund05] mettent en œuvre une étude de cas afin de vérifier comment réagit la technique du pare-feu lorsque appliquée à un gros système. Rothermel et al. [Rothermel00] présentent une

technique de sélection des cas de test basée sur la technique présentée précédemment [Rothermel97-1]. La technique fonctionne selon le même principe, mais elle est étendue afin de prendre en compte les spécificités du paradigme objet. La technique de sélection de Harrold et al [Harrold01] est la première qui supporte le langage Java et les constructions orientées objets de ce dernier. Seul l'aspect de la sélection des cas de test est abordé dans le cadre de cette technique. Les trois étapes principales suivantes permettent la sélection des cas de tests:

a. Construction d'un graphe qui représente le flot de contrôle et les informations relatives au type des classes;
b. Une traversée du graphe de contrôle permet d'identifier les arcs dangereux;
c. En se basant sur la matrice de couverture obtenue par l'instrumentation du code, les cas de test de la suite du programme original qui couvrent un arc dangereux sont choisis.

Cette technique utilise l'analyse statique sous forme d'analyse du code et utilise aussi l'analyse dynamique grâce à l'instrumentation du code. Le système de sélection des cas de test de régression qui a été crée pour implémenter cette méthode est RETEST. Ce dernier est fait de trois composants principaux :

a. Un analyseur dynamique (Profiler);
b. Un analyseur statique (DejaVOO, extension de DejaVu de Rothermel et al. [Rothermel04]);
c. Un sélecteur de tests (SelectTests).

Wu et al. [Wu99] proposent une technique qui se base sur les relations de dépendance entre les fonctions pour identifier les variables, les fonctions et les relations de dépendance des fonctions qui sont affectées par la modification. Un cas de test va être choisi seulement s'il couvre une fonction modifiée qui a un impact sur le comportement du système. Afin de déterminer si une fonction modifiée a un impact sur le comportement du système, on crée un graphe d'appels de fonctions pour chaque cas de test. Les relations entre les fonctions sont basées sur les associations entre les données membre et les relations entre les appels de

fonction et les retours d'appel. Li et al. [Li99] présentent un survol de quelques approches de test de régression.

### 1.3 Les approches basées sur les modèles

Pilskalns et al. [Pilskalns06] présentent une approche qui permet d'effectuer la sélection des tests de régression grâce à l'analyse des diagrammes de classes et des diagrammes de séquences tout en prenant en compte les expressions OCL (Object Constraint Language). L'approche consiste à combiner les informations relatives aux diagrammes de classes et aux diagrammes de séquences dans un graphe dirigé acyclique. La figure 2 illustre un exemple de ce graphe.

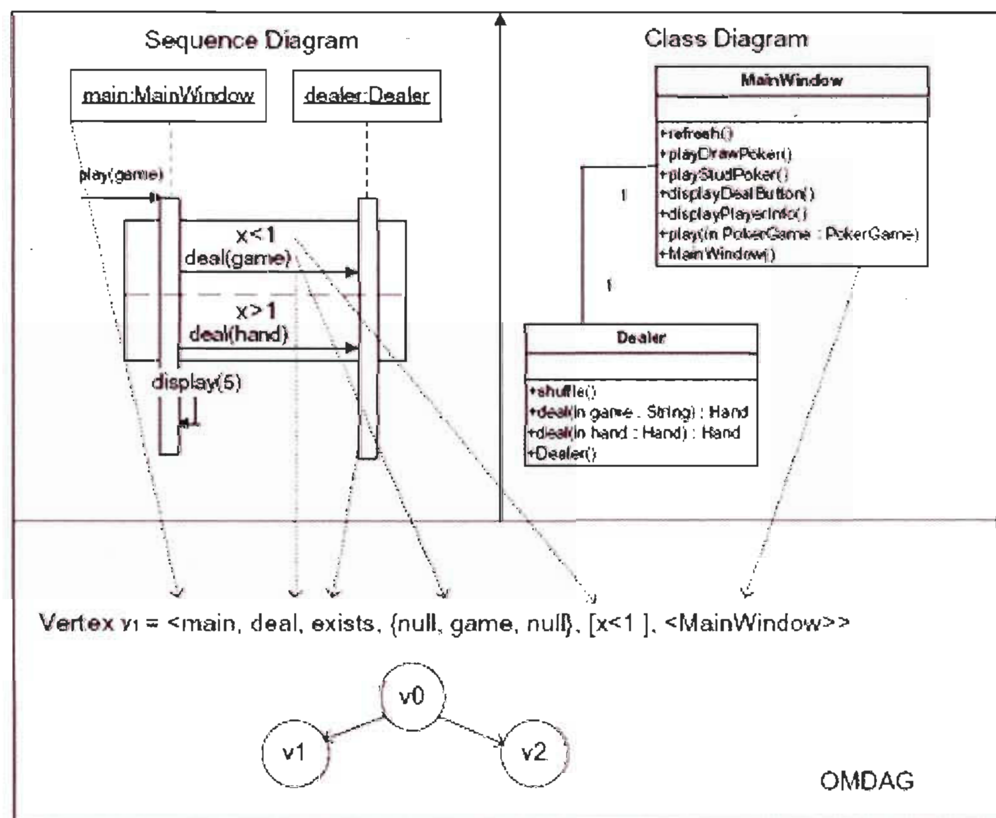


Figure 2 Exemple de génération du graphe acyclique.

Source : Pilskalns et al. [Pilskalns06].

Dans cette figure, on peut voir comment sont traduits les scénarios des diagrammes de séquences en vecteurs, chaque vecteur étant un chemin d'exécution possible. Les cas de tests générés à partir de ces séquences assignent différentes valeurs aux variables qui se trouvent dans les nœuds conditionnels. Dans la figure précédente, par exemple, le nœud v1 a comme condition que x doit être plus petit que 1. Cette condition agit comme une garde : afin de pouvoir traverser l'arête qui va de v0 à v1, la condition doit être satisfaite. Il est à noter que les contraintes formelles exprimées en OCL (Object Constraint Language) entrent aussi en compte pour détecter des incohérences.

Briand et al. [Briand02] présentent une technique qui permet d'évaluer tôt dans le processus l'importance de l'impact des changements. Les modèles étudiés sont les diagrammes de classes, les diagrammes de séquences et les diagrammes de cas d'utilisations. Les deux versions des différents types de modèles sont comparées et les cas de test se voient attribués une catégorie :

a. Désuet : Un cas de test est désuet si la séquence d'exécution n'est plus valide.
b. Re-testable : Le cas de test demeure valide mais une ou plusieurs méthodes peuvent avoir changées.
c. Réutilisable : Le cas de test demeure valide et aucun changement n'a été apporté.

Par la suite, les cas de test classifiés re-testables doivent être exécutés. Xu et al. [Xu03] présentent une technique de sélection des cas de test de régression basée sur les spécifications. Les cas de test sont générés à partir de la spécification formelle. L'idée générale des auteurs est d'utiliser la différence entre les spécifications de P et de P' (version modifiée) plutôt que d'utiliser seulement la suite de tests originale. Les deux spécifications sont traitées par un comparateur qui identifie les différences grâce à différents critères de couverture. Ensuite, chaque propriété extraite couvre un chemin de la spécification qui a été modifiée, lequel on souhaite re-tester. Ensuite, la spécification P' passe une vérification formelle de modèle. Chaque propriété qui est vérifiée formellement est considérée comme

n'ayant pas été affectée. Chaque propriété qui échoue lors de la vérification formelle est considérée comme ayant été affectée et doit donc être re-testée. Les cas de test sont générés à partir de chaque contre-exemple. Korel et al. [Korel02] présentent une approche utilisant le langage de description formelle EFSM, soit Machine à États Finis Étendue. Un modèle EFSM contient des états et des transitions. Les différences entre le modèle original et le modèle modifié sont extraites et représentées sous forme de modifications élémentaires (ajout d'une transition, retrait d'une transition). Pour chaque modification élémentaire, une stratégie de réduction est utilisée pour éliminer les tests répétitifs. Un graphe de dépendance EFSM est généré. Ce graphe capture les dépendances de données et de contrôle. Ces dépendances représentent les interactions entre les transitions du modèle.

## 1.4 Conclusion

Plusieurs techniques portant sur les tests de régression ont été développées dans le passé et une revue de la littérature en résume les principaux points. Le chapitre 2 porte sur les critères utilisés pour évaluer les techniques de tests de régression. Une revue de la littérature permet de synthétiser les différents critères d'évaluation.

# CHAPITRE 2

# CRITÈRES D'ÉVALUATION

Afin d'évaluer l'approche proposée de façon empirique, des critères d'évaluation doivent être définis. Ces derniers permettront d'évaluer les résultats obtenus sur l'étude de cas considérée d'une part, et de situer l'approche proposée par rapport à certaines approches existantes d'autre part. Dans un premier temps, il est essentiel de synthétiser les différentes façons adoptées dans la littérature afin d'évaluer les approches supportant les tests de régression. Par la suite, nous donnerons les bases de la méthode d'évaluation qui sera utilisée dans le cadre de notre étude.

## 2.1 État de l'art

L'approche de Gupta et al. [Gupta92] est basée sur les flots de données. Par conséquent, elle est évaluée par rapport au nombre d'associations de définition/utilisation des variables qui sont affectées par une modification. Le même principe est utilisé pour évaluer l'approche de Agrawal et al. [Agrawal93]. Chen et al. [Chen94] se basent sur le nombre de cas de tests sélectionnés en comparaison avec le nombre de cas de tests total. Korel et al. [Korel98] évaluent leur méthode en fonction du temps d'exécution. Si le temps d'exécution de la technique de sélection des cas de test et l'exécution des tests combinés est plus rapide que de tout re-tester, alors la méthode est jugée efficace. Rothermel et al. [Rothermel94, Rothermel96, Rothermel02] présentent une plateforme qui permet d'évaluer les diverses approches existantes. Quatre critères sont définis dans le cadre de ces recherches. Premièrement, l'inclusivité représente le nombre (en pourcentage) de cas de tests sélectionnés parmi tous ceux qui sont susceptibles de révéler des fautes. Ce critère sert à démontrer si une approche est de type sécuritaire, c'est-à-dire que tous les cas de tests qui sont susceptibles de révéler des fautes doivent être sélectionnés. Ensuite, la précision représente le nombre (en pourcentage) de cas de tests qui ne sont pas susceptibles de révéler des fautes mais qui sont sélectionnés quand même. Troisièmement, l'efficacité est évaluée

en deux volets. L'efficacité en termes d'espace représente la quantité d'informations qui doivent être conservées afin d'effectuer l'analyse. L'efficacité en termes de temps consiste à déterminer si la sélection des cas de tests (l'analyse et l'exécution) est plus rapide que de tout re-tester. Finalement, le dernier critère est plutôt qualitatif. Il s'agit de la généralité, c'est-à-dire vérifier si l'approche peut être appliquée à toutes les constructions d'un langage donné et si l'approche est limitée au niveau de la portée (certaines approches ne permettent que de traiter le corps des méthodes alors que d'autres sont capables de traiter un programme en entier). Bible et al. [Bible01] utilisent deux critères afin d'évaluer leur approche. Premièrement, la précision est évaluée en fonction du nombre de cas de tests qui ne doivent pas être re-testés. Finalement, l'efficacité est évaluée selon un modèle de coûts. Le coût $A_m$ d'appliquer une technique de sélection M et le coût de l'exécution et de la validation des cas de tests sélectionnés $T_m$ pour P' doit être inférieur au temps de l'exécution et de la validation de la suite de tests originale (tout re-tester). Rothermel et al. présentent dans [Rothermel00] leur première approche de sélection de cas de tests pour le paradigme orienté objet. L'approche est évaluée en fonction du nombre de cas de tests sélectionnés selon deux niveaux de granularité différents. Graves et al. [Graves98] évaluent leur approche selon les même critères que Bible et al. [Bible01], c'est-à-dire la précision et l'efficacité. Kim et al. [Kim00] utilisent deux critères afin d'évaluer leur approche : les économies et les coûts. Les économies correspondent à la réduction de la suite de test et les coûts correspondent au temps requis pour effectuer la sélection des cas de tests. Cette façon d'évaluer est très apparentée à celle de Bible et al. [Bible01] mais en utilisant des termes différents. Xu et al. [Xu03], Ren et al. [Ren05], Wu et al. [Wu99] et Korel et al. [Korel02] utilisent le critère de la réduction de la suite de tests afin d'évaluer leurs méthodes. Il s'agit de comparer le nombre de cas de tests sélectionnés et le nombre de cas de tests total. Harrold et al. [Harrold01] présentent la première technique de sélection des cas de tests applicable au langage Java. Le premier critère utilisé porte sur la réduction de la suite de tests, soit le nombre de cas de tests sélectionnés comparativement au nombre total de cas de tests. Le deuxième critère concerne le niveau de granularité de l'approche. Il s'agit du pourcentage de cas de tests sélectionnés en fonction du niveau de granularité. Briand et al. [Briand02] introduisent un nouveau concept concernant la réutilisation de la suite de tests.

En effet, suite à l'analyse et à la sélection des cas de tests, ces derniers sont classés selon trois catégories :

a. Obsolète : Les cas de tests qui n'ont plus de raison d'être (par exemple, si une méthode est supprimée);
b. Re-testable : Ce sont les cas de tests sélectionnés lors de l'analyse. Ces cas doivent être impérativement exécutés;
c. Réutilisable : Ces cas n'ont pas été sélectionnés, donc ils ne seront pas exécutés. Par contre, ils pourraient être réutilisés lors d'une prochaine itération.

Afin de déterminer la réduction de la suite de tests, il suffit de comparer le nombre de cas identifiés comme re-testables au nombre total de cas de tests. Pilskalns et al. [Pilskalns06] réutilisent la méthode de classification et d'évaluation de Briand et al. [Briand02]. Kung et al. [Kung93] introduisent la notion de pare-feu pour la sélection des cas de tests. L'analyse permet de déterminer les méthodes modifiées ainsi que les méthodes affectées par les modifications. Ainsi, tous les cas de tests qui couvrent ces méthodes sont sélectionnées. La réduction de la suite de tests est ensuite facilement obtenue en comparant avec le nombre total de cas de tests. White et al. [White05] étendent la technique du pare-feu vue précédemment. Les critères utilisés pour évaluer la méthode sont le nombre de cas de tests sélectionnés ainsi que l'effort requis en terme de temps d'exécution.

## 2.2 Observations

En étudiant les diverses techniques d'évaluation de façon chronologique, certaines tendances se dégagent. Les premières techniques de sélection des cas de tests avaient pour objectif de démontrer qu'elles étaient rentables et avantageuses par rapport à la stratégie de re-test complet, surtout au niveau du temps et de l'effort requis. Par la suite, la rentabilité des techniques de sélection étant devenue un acquis, et avec l'émergence de techniques s'appliquant au paradigme orienté objet, de nouvelles préoccupations sont apparues. Les plus récentes techniques se basent sur la réduction de la suite de tests pour leur évaluation.

De plus, certains travaux ont introduit la réutilisation de la suite de tests comme élément central d'évaluation. La table III illustre un condensé des différentes techniques d'évaluation décrites précédemment.

| Approche | Méthode d'évaluation |
|---|---|
| [Gupt92], [Agrawal93] | Nombre d'associations de définition/utilisation des variables qui sont affectées par une modification. |
| [Korel98] | Temps d'exécution comparativement à tout re-tester. |
| [Rothermel94], [Rothermel96], [Rothermel02] | Inclusivité : % de cas de test sélectionnés parmi tous ceux qui sont susceptibles de révéler des fautes.<br>Précision : % de cas de tests qui ne sont pas susceptibles de révéler des fautes mais qui sont sélectionnés quand même.<br>Efficacité : Temps d'exécution inférieur à tout re-tester.<br>Généralité : Applicable à toutes les constructions du langage. |
| [Bible01], [Graves98], [Kim00], [White05] | Précision : Nombre de cas de tests éliminés.<br>Modèle de coûts : Temps d'exécution inférieur à tout re-tester. |
| [Rothermel06], [Harrold01] | Nombre de cas de tests sélectionnés selon deux niveaux de granularité. |
| [Chen94], [Xu03], [Ren05], [Wu99], [Korel02], [Kung96] | Nombre de cas de tests sélectionnés comparativement au nombre de cas de tests total. |
| [Briand02], [Pilskalns06] | Réutilisation de la suite de test et nombre de cas de tests sélectionnés |

Table III Condensé des différentes techniques d'évaluation décrites dans l'état de l'art.

## 2.3 Description de la méthode d'évaluation choisie

Dans le cadre de l'approche adoptée, les préoccupations adressées sont essentiellement la réduction de la suite de tests ainsi que sa réutilisation. L'objectif étant de démontrer les gains effectués en termes de temps et en termes de réduction de l'effort de test nécessaire. Nous donnons, dans ce qui suit, les définitions des critères qui sont à la base de la méthode d'évaluation retenue dans le cadre de notre étude.

**Définition 1** : *Soit $RED(P)=C_i/C_n$ où P représente un programme donné, $C_i$ représente les séquences sélectionnées et $C_n$ représente l'ensemble de toutes les séquences du programme. Le résultat de la fonction RED(P) représente la réduction de la suite de tests. La réduction de la suite de tests est déterminée en fonction du nombre de séquences de tests qui sont considérées (seront re-testées) par rapport au nombre total de séquences de tests.*

**Définition 2** : *Soit $REU(P)=CT_i/CT_n$ où P représente un programme donné, $CT_i$ représente le nombre de cas de tests qui peuvent être réutilisés et $CT_n$ représente le nombre total de cas de tests. REU(P) représente la réutilisation de la suite de tests. La réutilisation de la suite de tests est déterminée en fonction du taux de réutilisation au sein des séquences de tests sélectionnées. Pour chaque séquence de tests sélectionnée, une comparaison est faite avec la séquence correspondante (si applicable) de la version i du logiciel. Le taux de réutilisation local est le nombre de méthodes qui faisaient partie de la séquence correspondante de la version i comparativement au nombre total de méthodes dans la séquence de la version i+1. Pour obtenir le taux de réutilisation global, il suffit de l'appliquer à l'échelle de toutes les séquences sélectionnées.*

## 2.4 Conclusion

Une revue de la littérature permet d'identifier les différents critères d'évaluation utilisés par plusieurs approches portant sur les tests de régression. Une synthèse est illustrée à la table III. Enfin, deux critères d'évaluation sont définis pour évaluer la technique présentée dans ce mémoire. Le chapitre 3 traite de la comparaison de programmes, la technique utilisée pour identifier les différences entre deux versions successives d'un même logiciel.

# CHAPITRE 3

# COMPARAISON DE PROGRAMMES

## 3.1 Approche

Afin d'identifier les cas d'utilisation qui ont subit un changement lors d'une activité de maintenance du logiciel par exemple, la première étape consiste à comparer la version d'origine du programme et sa version modifiée afin d'identifier les méthodes qui ont subies des changements. Grâce à l'analyse de programmes, les différences entre les deux versions sont identifiées. Pour ce faire, l'analyse statique de programmes est utilisée. La notion d'analyse statique de programmes couvre une variété de méthodes utilisées pour obtenir des informations sur le comportement d'un programme lors de son exécution sans réellement l'exécuter [Wikipedia01]. L'analyse statique du programme est basée sur son analyse lexicale et syntaxique.

## 3.2 Analyse lexicale et syntaxique

L'analyse syntaxique consiste à exhiber la structure d'un texte, généralement un programme informatique ou du texte écrit dans une langue naturelle. Un analyseur syntaxique est un programme informatique qui réalise cette tâche. Cette opération suppose une formalisation du texte, qui est vu le plus souvent comme un élément d'un langage formel, défini par un ensemble de règles de syntaxe formant une grammaire formelle. La structure révélée par l'analyse donne alors précisément la façon dont les règles de syntaxe sont combinées dans le texte. Cette structure est souvent une hiérarchie de syntagmes, représentable par un arbre syntaxique dont les nœuds peuvent être décorés (dotés d'informations complémentaires). L'analyse syntaxique fait habituellement suite à une analyse lexicale qui découpe le texte en un flux de lexèmes, et sert à son tour de préalable à une analyse sémantique. Connaître la structure syntaxique d'un énoncé permet d'expliciter les relations de dépendance (par exemple entre sujet et objet) entre les différents lexèmes, puis de construire une

représentation du sens de cet énoncé [Wikipedia02]. L'outil de base utilisé dans le cadre de notre approche pour effectuer l'analyse syntaxique et lexicale est Javacc [Javacc01]. Ce dernier permet, à partir d'une grammaire Java, de générer les classes nécessaires pour effectuer l'analyse syntaxique et lexicale. Cet analyseur prend en entrée un code source Java et détermine s'il correspond formellement à la grammaire. L'outil a été étendu pour les besoins de notre approche.

### 3.3 Arbre syntaxique abstrait

L'arbre syntaxique complet généré suite à l'analyse lexicale et syntaxique contient beaucoup trop de nœuds et des branches qui n'affectent pas la sémantique du programme. Cela rendrait très complexe l'opération de comparaison. C'est pour cette raison que l'outil JJtree, qui est un module complémentaire de Javacc, permet la génération de l'arbre syntaxique abstrait. Communément appelé AST pour 'Abstract Syntax Tree', cet arbre diffère de l'arbre syntaxique complet par l'omission des nœuds et des branches qui n'affectent pas la sémantique du programme. Un exemple classique est l'omission des parenthèses de groupement, puisque dans un arbre syntaxique abstrait, le groupement des opérandes est explicite dans la structure de l'arbre [Wikipedia03]. Structurellement, les nœuds internes sont marqués par des opérateurs et les nœuds fils (feuilles ou nœuds externes) représentent les opérandes de ces opérateurs. La figure 3 illustre un exemple d'arbre syntaxique abstrait.

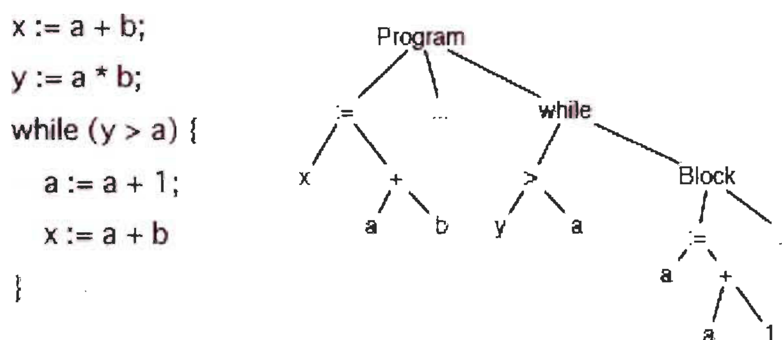


Figure 3 Exemple d'arbre syntaxique abstrait [CMSC01].

### 3.4 Processus de comparaison

Une première étape, dans le cadre de notre démarche, consiste à analyser les répertoires qui contiennent le code source pour les versions i et i+1. Les fichiers sont tous lus et analysés syntaxiquement. Les méthodes ajoutées et supprimées sont immédiatement ajoutées à la liste des méthodes modifiées. Les autres méthodes subissent une comparaison au niveau de leur arbre syntaxique abstrait. La moindre différence au niveau de l'arbre syntaxique abstrait démontre que la méthode en question a été modifiée. Ces méthodes sont ajoutées à l'ensemble M des méthodes modifiées. Un outil a été développé afin d'automatiser cette tâche complètement. Les détails relatifs au fonctionnement de l'outil en question se trouvent au chapitre 10.

### 3.5 Conclusion

Les fondements théoriques de la comparaison de programmes sont présentés et le processus de comparaison est défini. Le chapitre 4 décrit les différents diagrammes UML qui sont utilisés dans le cadre de l'approche et des exemples sont donnés.

# CHAPITRE 4

# MODÈLES UML

L'approche proposée est pilotée par les cas d'utilisation. L'objectif étant, suite à une activité de maintenance du système, d'identifier et de tester tous les cas d'utilisation ayant subis un changement. Pour y parvenir, les modèles UML sont utilisés. Le langage de modélisation UML (Unified Modeling Language) est un langage standardisé qui permet la spécification, la visualisation, la construction et la documentation des différents artéfacts d'un système logiciel. La notation graphique UML est la plus couramment utilisée pour représenter des systèmes logiciels. Afin d'obtenir de l'information sur les différents cas d'utilisation qui composent le système, il suffit d'analyser les diagrammes d'états de cas d'utilisation qui les décrivent. Par la suite, il est possible d'extraire le comportement dynamique de chaque cas d'utilisation en analysant, en complément aux diagrammes d'états, les diagrammes de collaboration pertinents. Finalement, les cas d'utilisation modifiés sont ceux qui comportent aux moins une méthode faisant partie de la liste des méthodes modifiées obtenue lors de l'analyse de programme. Parmi les avantages à utiliser les modèles UML dans le cadre de cette approche (en particulier pour la conception des cas de test), le plus important est sans aucun doute que l'approche est indépendante de tout langage de programmation.

## 4.1 Cas d'utilisation

Un cas d'utilisation est la description du comportement du système logiciel alors qu'il répond à une demande qui provient de l'extérieur du système. Les cas d'utilisation ne sont pas des constructions orientées objet, il ne s'agit que d'histoires écrites. Chaque cas est en fait une collection de scénarios de réussite ou d'échec qui décrit la façon dont un acteur utilise un système pour atteindre son but [Larman01]. La figure 4 illustre le cas d'utilisation 'traiter une vente' d'une application de gestion de ventes.

**Acteur Principal : le caissier**
**Parties prenantes et intérêts :**

**Le caissier** : Il veut un moyen de saisie exact et rapide et aucune erreur de paiement.
**Le vendeur** : Il veut que les commission sur les ventes soient mises à jour.
**Le client** : Il veut des achats et un service rapide avec le minimum d'efforts. Il veut également une preuve d'achat en cas de retour.
**L'entreprise** : Elle veut enregistrer correctement les transactions et satisfaire les intérêts des clients.

**Préconditions** :
Le caissier est identifié et authentifié.

**Postconditions** :
La vente est enregistrée. La taxe est correctement calculée. La comptabilité et l'inventaire son mis à jour. Le reçu est généré. Les autorisations de paiement sont enregistrées.

**Scénario principal :**

1. Le client arrive à la caisse avec des marchandises et/ou des services à acheter.
2. Le caissier démarre une nouvelle vente.
3. Le caissier entre le code de l'article.
4. Le système enregistre l'article et présente sa description, son prix et le total en cours. Le prix est calculé à partir d'un ensemble de règles.

Le caissier répète les étapes 3 et 4 jusqu'à ce que tous les articles soient passés.

5. Le système présente le total avec les taxes calculées.
6. Le caissier communique le total au client et en demande le paiement.
7. Le client paie et le système traite le paiement.
8. Le système enregistre la vente.
9. Le système présente un reçu
10. Le client part avec les marchandises et le reçu.

**Extensions (ou scénarios alternatifs) :**
(...)
3-6b. Le client demande au caissier d'annuler la vente

1. Le caissier annule la vente.

(...)

Figure 4 Cas d'utilisation 'Traiter une vente' [Larman01].

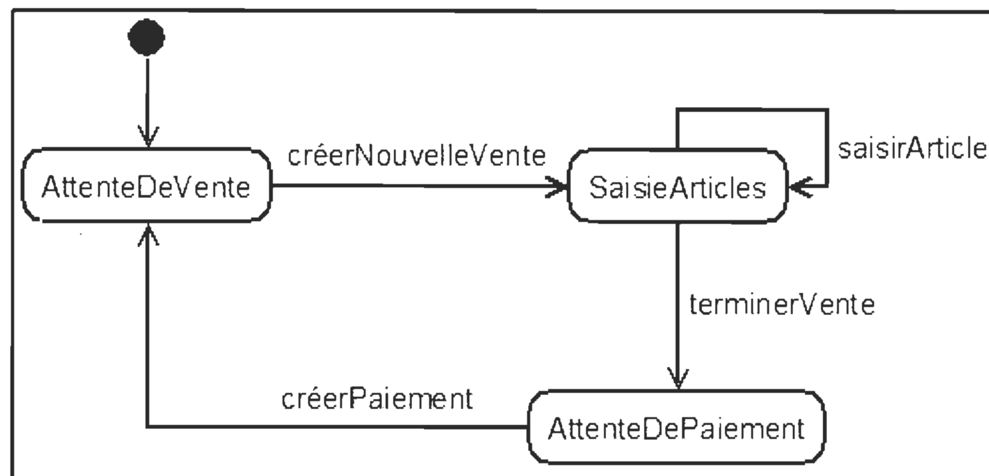


Figure 5 Diagramme d'états du cas d'utilisation 'traiter une vente' [Larman01].

### 4.2 Diagramme d'états de cas d'utilisation

Un diagramme d'états de cas d'utilisation est un diagramme d'états décrivant les successions valides d'événements externes reconnus et gérés par le système dans le contexte des cas d'utilisation. Un diagramme d'états se compose de deux éléments : les états et les transitions qui permettent de passer d'un état à un autre. La figure 5 illustre le diagramme d'états du cas d'utilisation 'traiter une vente'. Dans le cadre de l'approche, les diagrammes d'états de cas d'utilisation sont décrits à l'aide de la notation XML (Langage à balises extensible). À cet égard, un outil a été développé afin d'en automatiser l'analyse. Afin de déterminer si des modifications ont eu lieu, il suffit de vérifier les états et les transitions en comparant le diagramme d'états (description) de la version modifiée avec celui de la version précédente. L'analyse formelle est faite à l'aide de l'outil développé. Chaque transition possède un état source et un état destination. Il est alors possible de représenter les diagrammes sous forme de graphes et ainsi en parcourant les graphes toute différence peut être identifiée.

### 4.3 Diagramme de collaboration

Un diagramme de collaboration est l'un des diagrammes d'interaction UML qui sont utilisés pour illustrer comment les objets interagissent entre eux en échangeant des messages. Ils permettent de représenter le contexte d'une interaction, car on peut y préciser les états des objets qui interagissent. Le lien qui unit les diagrammes de collaboration aux cas d'utilisation est que le comportement du système pour chacun des scénarios accompagnant les cas d'utilisation est décrit par les divers diagrammes de collaboration [Galland02]. La dimension temporelle est ajoutée grâce à des numéros de séquence. Selon Larman [Larman01], les avantages du diagramme de collaboration résident dans l'économie en termes d'espace. Il est, par ailleurs, mieux adapté à l'illustration des branchements complexes, des itérations et des comportements concurrents. La figure 6 illustre le diagramme de collaboration de 'saisirArticle'. Tout comme les diagrammes d'états de cas d'utilisation, les diagrammes de collaboration sont décrits, dans le cadre de l'approche adoptée, selon la notation XML. Tel que spécifié précédemment, un outil a été crée afin d'en automatiser l'analyse.

Afin de déterminer si un diagramme de collaboration a subit une modification, il suffit de le représenter sous forme d'un graphe et de suivre l'ordonnancement des messages. Si une signature de méthode est modifiée, si un changement d'ordonnancement est détecté, ou si une des méthodes qui compose le diagramme fait partie de l'ensemble M des méthodes ayant subies une ou plusieurs modifications, alors le diagramme de collaboration est marqué comme étant modifié.

### 4.4 Liens entre le diagramme d'états de cas d'utilisation et le diagramme de collaboration

Une transition étant une interaction entre deux objets. Un diagramme de collaboration peut être lié à cette transition pour autant que la méthode en question soit minimalement

complexe pour justifier la création d'un diagramme. Dans l'exemple présenté, le diagramme d'états de cas d'utilisation pour le cas 'traiter une vente' comporte une transition qui se nomme 'saisirArticle' (Figure 5). Maintenant, en observant le diagramme de collaboration de 'saisirArticle' (Figure 6) il est possible d'établir le lien. Pour établir ce lien, il faut vérifier si les signatures de la méthode correspondent.

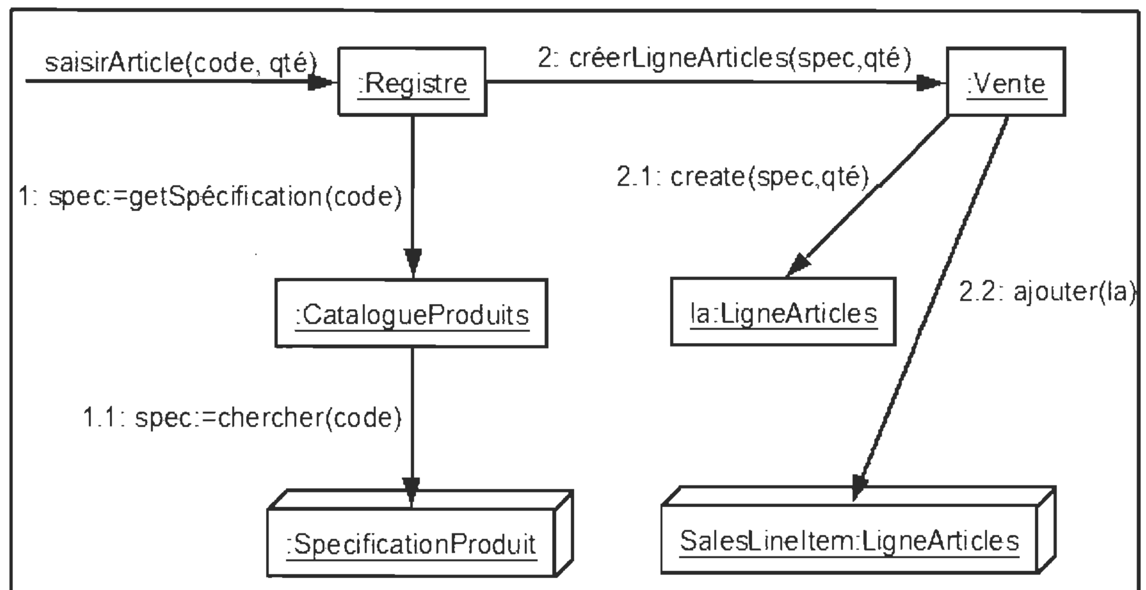


Figure 6 Diagramme de collaboration pour le scénario 'saisirArticle' [Larman01].

### 4.5 Conclusion

Les différents modèles UML utilisés dans le cadre de cette approche sont présentés et un exemple est donné pour chacun. De plus, les liens entre les différents modèles sont illustrés grâce aux exemples. Le chapitre 5 présente la méthodologie de l'approche.

# CHAPITRE 5

# MÉTHODOLOGIE DE L'APPROCHE

## 5.1 Méthodologie de l'approche

L'approche proposée est basée sur les cas d'utilisation et plus précisément sur les modèles UML qui les décrivent : diagrammes d'états de cas d'utilisations et diagrammes d'interactions. Elle permet d'identifier les cas d'utilisations ayant subit une modification et par conséquent les parties d'un système qui doivent être re-testées. Elle permet, par ailleurs, la sélection des cas de tests appropriés (parmi les tests existants) ainsi que la génération des nouveaux cas de tests (extension de la batterie existante). Étant basée sur les modèles, l'approche est indépendante d'un quelconque langage de programmation.

Soient deux versions d'un programme P : $V_1$ et $V_2$. La version $V_2$ est obtenue suite à des changements apportés à la version $V_1$. Les changements peuvent être de différents types (ajout, suppression et modification). Ces changements peuvent être effectués pour différentes raisons. Ils peuvent aussi concerner des fonctionnalités de haut niveau (cas d'utilisation). L'approche qui suit est à généraliser lors du passage d'une version $V_i$ à une autre $V_{i+1}$ d'un même programme P.

La première étape de l'approche consiste à procéder à une comparaison des deux versions du programme pour identifier l'ensemble M des méthodes modifiées. La comparaison des deux versions du programme se fait, comme mentionné précédemment au chapitre 3, par analyse statique du code des deux versions du programme. Cet ensemble peut contenir les méthodes existantes qui ont été modifiées, des méthodes qui existaient dans la première version et qui ont été supprimées, comme il peut aussi contenir de nouvelles méthodes.

La seconde étape consiste à identifier, parmi l'ensemble des méthodes M précédemment définies, l'ensemble $MC_i$ des opérations (méthodes) rattachées à chaque cas d'utilisation

$CU_i$. Cette seconde étape sera basée sur une analyse des diagrammes d'états-transitions décrivant le comportement des cas d'utilisations dans le cas des deux versions $V_1$ et $V_2$. Les diagrammes d'états-transitions dans le cadre de l'approche adoptée sont décrits comme mentionnés précédemment en XML. Cette analyse permettra surtout d'identifier les nouveaux ajouts en termes d'opérations. Cette analyse permettra d'identifier l'ensemble M(CU) des cas d'utilisations ayant subit une ou plusieurs modifications.

La troisième étape consiste en la génération des séquences de tests pour chaque cas d'utilisation modifié. Chaque séquence de tests correspond à un scénario particulier du logiciel. Les séquences générées couvrent le scénario de base du cas d'utilisation ainsi que ses différentes extensions. Cette étape se compose de deux phases : séquences réduites et séquences complètes.

Afin de mieux visualiser le processus, un exemple théorique est présenté. La figure 7 illustre le diagramme d'états d'un cas d'utilisation fictif (partie gauche) ainsi que le diagramme de collaboration de la méthode fictive M3 (partie droite).

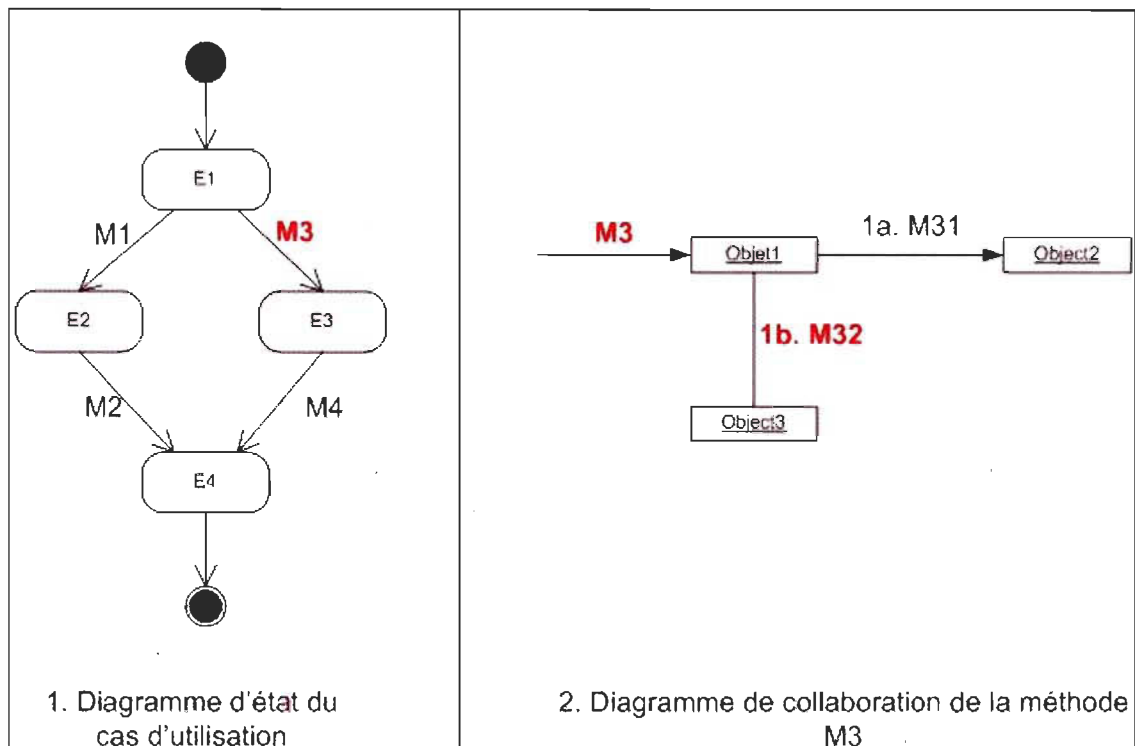


Figure 7 Exemple théorique de la méthodologie de l'approche.

Supposons que lors de la première étape, il a été déterminé que la méthode M32 de l'objet 'Objet3' a subit une modification; la méthode M32 fait partie de la collaboration de la méthode M3, en conséquence la méthode M3 est identifiée comme impactée par la modification. Comme la méthode M3 fait partie du cas d'utilisation fictif, ce dernier est considéré comme impacté et est ajouté à l'ensemble M(CU) des cas d'utilisations ayant subit un changement.

Suite à l'identification de l'ensemble M(CU), les chemins (qui correspondent aux différents scénarios) doivent être générés. Premièrement, au niveau du diagramme de collaboration de M3, il y a deux chemins possibles: M31 et M32. Étant donné que la méthode M31 n'a pas subit de changement, ce chemin n'est pas considéré (ne devra pas être re-testé). Par la suite, au niveau du diagramme d'états du cas d'utilisation, il y a deux chemins possibles: M1-M2 et

M3-M4. Étant donné que M1 et M2 n'ont pas subit de changement, ce chemin ne sera pas considéré. Enfin, au niveau du chemin M3-M4, la méthode M3 sera remplacée par son propre chemin (qui est M32, suite à l'analyse du diagramme de collaboration de la méthode M3) afin d'obtenir les chemins complets. Le chemin final de test est donc: M32-M4. Le résultat net dans ce cas est que plutôt de tester les quatre chemins du cas d'utilisation (en combinant les chemins du diagramme d'états avec les chemins de la méthode M3), seulement un chemin doit être re-testé.

Suite à la génération et la sélection des chemins à re-tester, les cas de tests sont sélectionnés (réutilisés) à partir de la batterie existante. La dernière étape consiste à générer les cas de tests pour les nouveaux chemins. Ces nouveaux cas de tests et ces nouveaux chemins sont ajoutés à la batterie de tests(mise à jour incrémentale de la batterie de tests au fur et à mesure des opérations de modification).

La figure 8 illustre le modèle général de la méthodologie proposée.

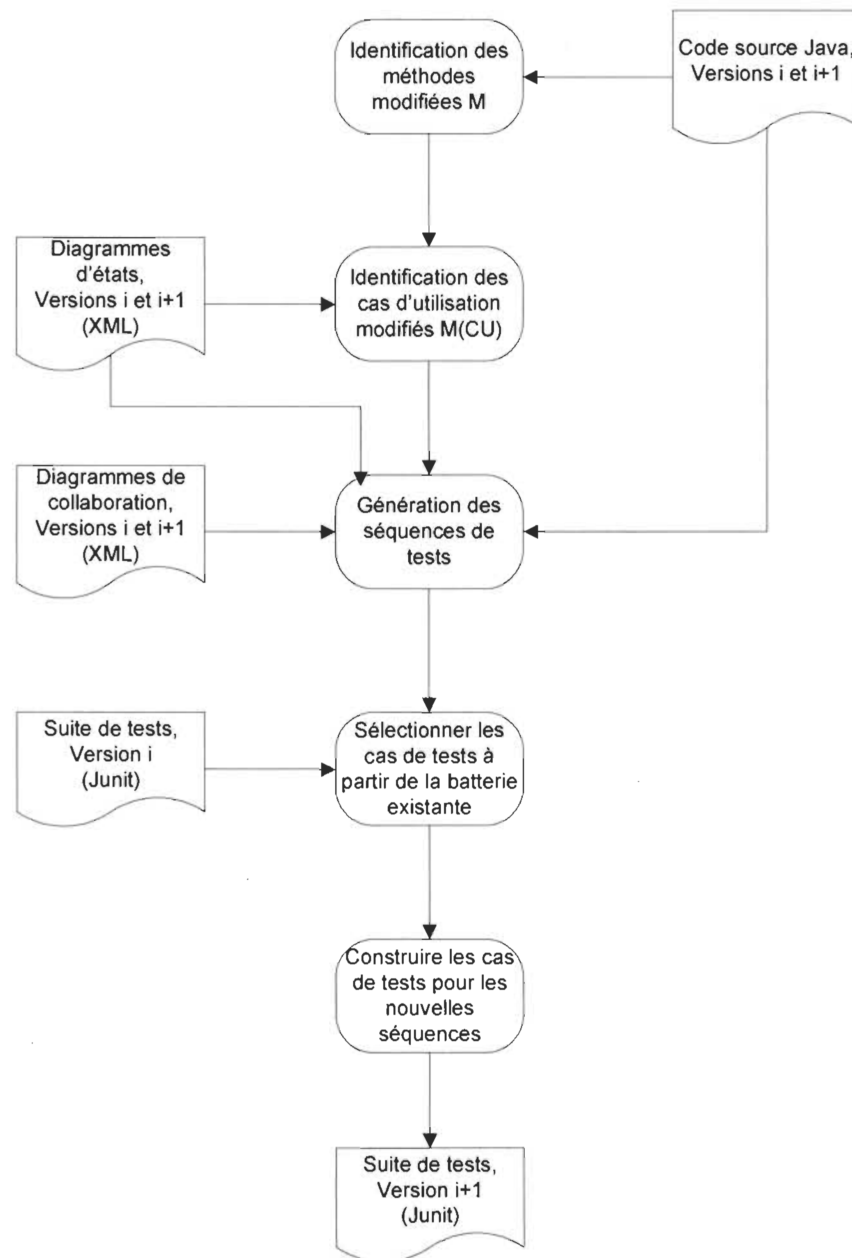


Figure 8 Modèle général de la méthodologie proposée.

## 5.2 Conclusion

La méthodologie de l'approche est présentée et un modèle général est illustré. Le chapitre 6 présente l'algorithme de génération des séquences de tests ainsi qu'un exemple théorique complet qui permet de mieux comprendre l'algorithme. Par la suite, la génération automatique des cas de tests est présentée.

# CHAPITRE 6

# GÉNÉRATION DES SÉQUENCES DE TESTS

## 6.1 Introduction

Nous abordons, dans le cadre de ce chapitre, la génération des séquences de tests pour les cas d'utilisation ayant subit une ou plusieurs modifications. Tel que présenté au chapitre précédent, un programme contient plusieurs scénarios d'exécution. Ces scénarios sont déterminés, comme présenté précédemment, par analyse des diagrammes d'états de cas d'utilisation et des diagrammes de collaboration. Une séquence de tests, qui est en fait une suite d'appels de méthodes, correspond à un scénario d'exécution du programme. L'objectif principal du test de régression étant de s'assurer que, suite aux modifications apportées, de nouvelles erreurs n'ont pas été introduites et qu'il n'y ait pas eu de régression dans le système. Il suffit, dans ce contexte, de tester tous les scénarios qui sont affectés par au moins une modification. De cette façon, une couverture sécuritaire des sections modifiées (ou impactées par une modification) du logiciel est assurée. Dans ce qui suit, nous présenterons dans un premier lieu quelques concepts théoriques. Ces concepts sont reliés aux éléments de base de l'approche adoptée (graphe de contrôle réduit aux appels, passage à l'arbre des messages et génération des séquences de test). Nous présenterons par la suite le processus de génération des séquences de tests. Le cas d'utilisation « traiter une vente », présenté dans les chapitres précédents, est ensuite utilisé comme exemple théorique.

## 6.2 Concepts théoriques

### 6.2.1 Graphe de contrôle réduit aux appels

Les graphes de contrôle réduits aux appels (CCG) permettent de synthétiser le comportement du programme. Ils précisent les enchaînements lors de l'exécution des appels

[Badri05]. Les deux définitions qui suivent concernent respectivement le graphe de contrôle et le graphe de contrôle réduit aux appels :

**Définition 3** : *Un graphe de flot de contrôle est un graphe orienté. Les nœuds de ce graphe représentent soit des points de décision (« if-then-else, while, case »), une instruction ou un bloc séquentiel d'instructions. Un bloc séquentiel d'instructions est une séquence d'instructions telles que si nous exécutons la première instruction, nous sommes sûrs d'exécuter les autres, et toujours dans le même sens. Un arc orienté lie un nœud* $N_i$ *à un noeud* $N_j$ *s'il est possible d'exécuter l'instruction correspondante à* $N_j$ *immédiatement après celle associée au nœud* $N_i$. *Les arcs du graphe indiquent le transfert de contrôle d'un nœud à l'autre [Badri05].*

**Définition 4** : *Un graphe de contrôle réduit aux appels est un graphe de flot de contrôle dans lequel les nœuds, représentant les instructions ne conduisant pas à des appels, sont éliminés [Badri05].*

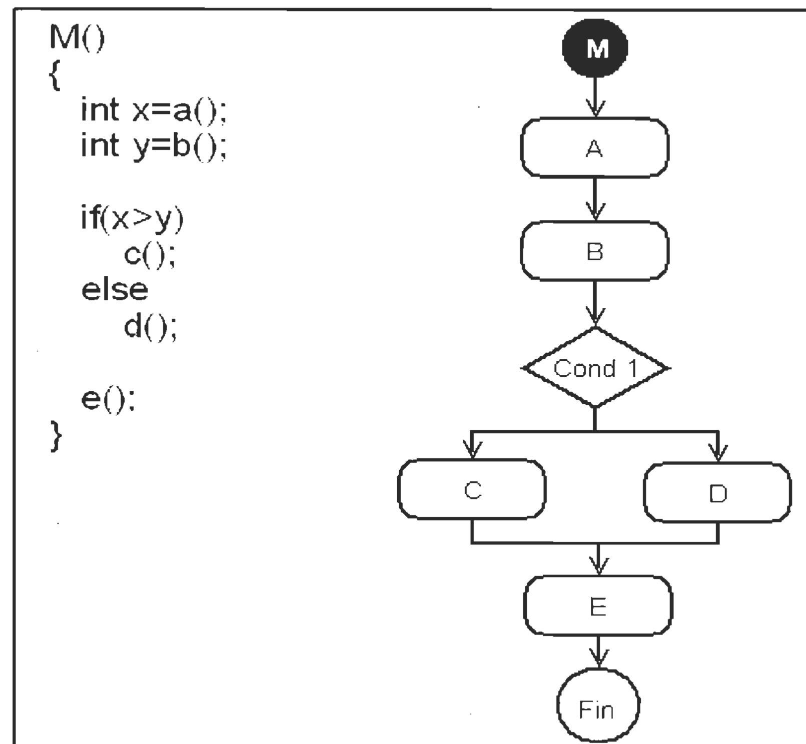


Figure 9 Exemple de graphe de contrôle réduit aux appels.

La figure 9 présente un graphe de contrôle réduit aux appels correspondant au pseudo code de la méthode M. En analysant le graphe, deux chemins d'exécution possibles sont identifiés pour la méthode M : ABCE et ABDE.

### 6.2.2 Arbre des messages

Le graphe de contrôle réduit aux appels est ensuite utilisé afin de générer un arbre des messages. En représentant les scénarios sous forme d'arborescence, il suffit de parcourir l'arbre de la racine jusqu'à chaque feuille en utilisant tous les chemins possibles afin d'obtenir la liste des séquences. La figure 10 illustre une transition complète à partir d'un pseudo code jusqu'à l'arbre des messages correspondant pour la méthode M. En parcourant l'arbre de la racine jusqu'à chaque feuille, on observe que deux séquences peuvent être identifiées : ABCE et ABDE.

### 6.2.3 Séquence principale

Suite à la génération d'un arbre des messages, ce dernier est convertit en une séquence particulière appelée séquence principale ou séquence compactée. Il s'agit d'une expression régulière qui permet de représenter la liste de tous les chemins d'exécution possibles de façon très compacte. La notation utilisée est la suivante : le symbole {séquence} signifie 0 ou plusieurs exécutions de la séquence. Le symbole (séquence1/séquence2) représente une exclusion mutuelle, c'est-à-dire qu'une seule des deux séquences peut être sélectionnée à la fois. Finalement, la notation [séquence] signifie que la séquence peut être exécutée ou non [Massicotte06]. Par exemple, si on considère l'exemple de la méthode M de la figure 9 vue précédemment, la séquence principale qui y correspond est la suivante : A, B, (C/D), E.

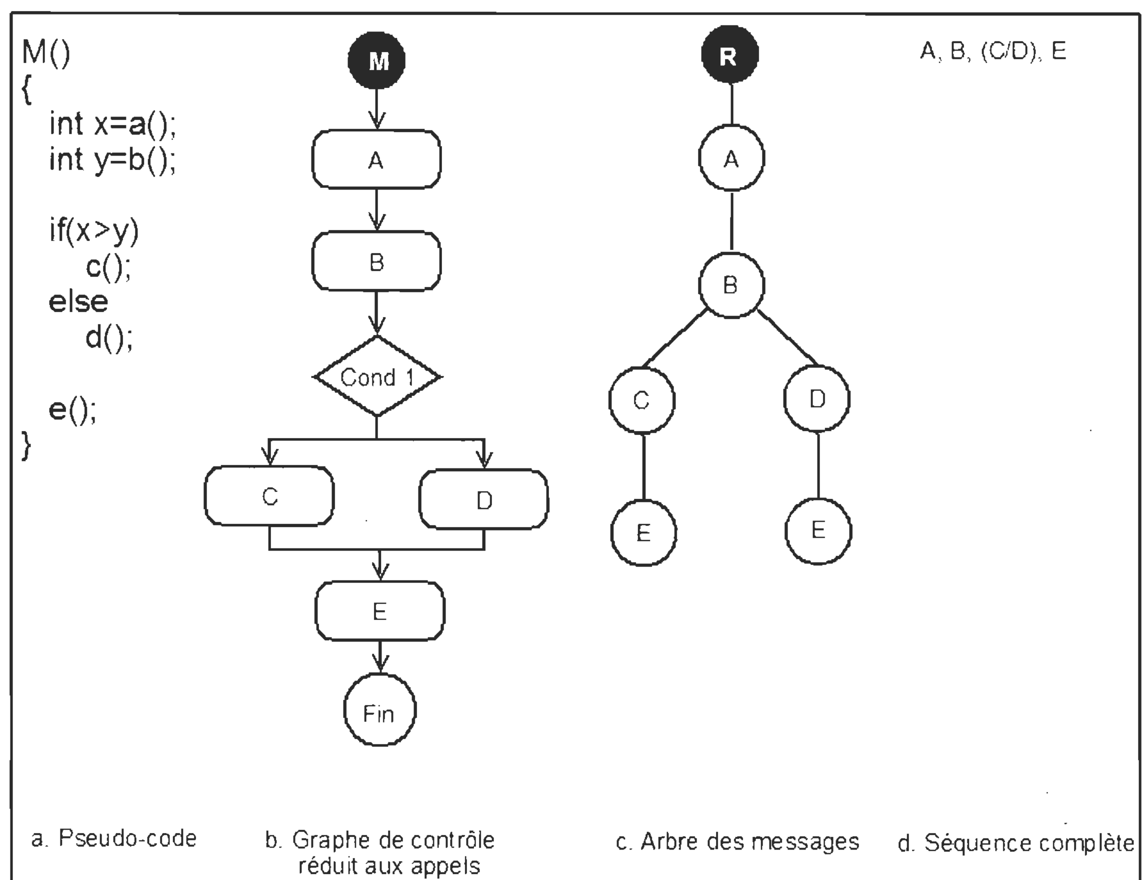


Figure 10 Exemple de transition allant d'un pseudo code jusqu'à l'arbre des messages

## 6.3 Processus de génération des séquences de tests

Plusieurs étapes sont impliquées dans le processus de génération des séquences de tests. Afin de faciliter la compréhension et faire le lien entre les étapes précédentes et la génération des séquences de tests, nous faisons dans ce qui suit quelques rappels importants. Plusieurs étapes préliminaires ont été vues au cours des chapitres précédents. Premièrement, l'ensemble M des méthodes modifiées a été identifié. Ensuite, l'ensemble $MC_i$ des opérations (méthodes) rattachées à chaque cas d'utilisation $CU_i$ a été identifié. Ces étapes ont permis d'identifier l'ensemble M(CU) des cas d'utilisation ayant subi une ou plusieurs modifications. Avec toutes ces informations, le processus de génération des séquences de test peut démarrer. Les étapes principales sont décrites par l'algorithme suivant:

Pour chaque cas d'utilisation $CU_i$ appartenant à l'ensemble M(CU) :

a. Pour chaque opération $op_i$ modifiée appartenant à l'ensemble $MC_i$ :

    a. Générer le graphe de contrôle réduit aux appels correspondant à partir d'une analyse du diagramme de collaboration (décrit en XML) de $op_i$;
    b. Construire l'arbre des messages complet correspondant;
    c. Générer la séquence principale pour la collaboration;
    d. Générer les séquences de tests à partir de la séquence principale (séquences réduites, opération en question);
    e. Sélectionner les cas de test à partir de la batterie existante;
    f. Construire des cas de tests pour les nouvelles séquences.

b. Générer le graphe de contrôle réduit aux appels du cas d'utilisation à partir d'une analyse du diagramme d'états de cas d'utilisation;
c. Construire l'arbre des messages en marquant les chemins modifiés, par analyse des deux diagrammes;
d. Générer les séquences de tests correspondant aux chemins modifiés;
e. Ces séquences seront dans un second temps étendues par intégration (aux bons endroits) des séquences réduites. Cette procédure donnera les séquences complètes de tests à appliquer;
f. Sélectionner les cas de test à partir de la batterie existante (voir chapitre suivant);
g. Construire les cas de test pour les nouvelles séquences (voir chapitre suivant).

Afin de mieux visualiser la récursivité impliquée dans le processus, un exemple théorique est proposé. Il s'agit du cas d'utilisation 'traiter une vente', vu au chapitre précédent.

## 6.4 Exemple théorique

Supposons que lors de la première étape, il ait été déterminé que la méthode 'créerLigneArticles' de la classe 'Registre' a subi une modification; cette méthode fait partie de la collaboration de la méthode 'saisirArticle' illustrée à la figure 11. Cela a comme conséquence que la méthode 'saisirArticle' est considérée comme impactée et devra être considérée. L'étape suivante consiste à générer le graphe de contrôle réduit aux appels correspondant à la collaboration. Le graphe de contrôle en question est illustré à section 'a' de la figure 12.

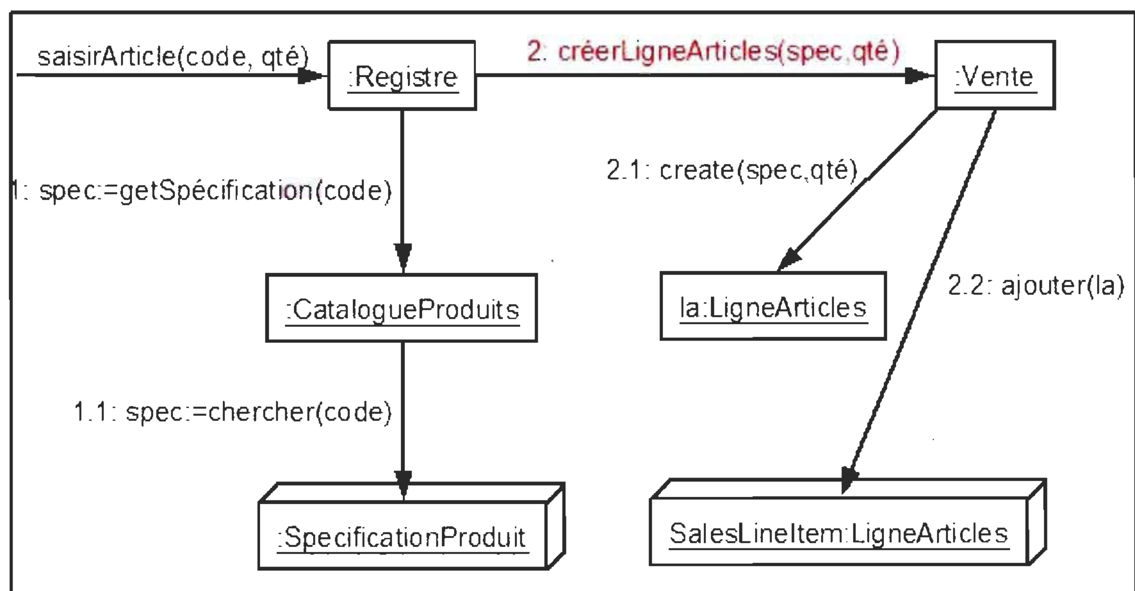


Figure 11 Diagramme de collaboration pour la méthode 'saisirArticle'.

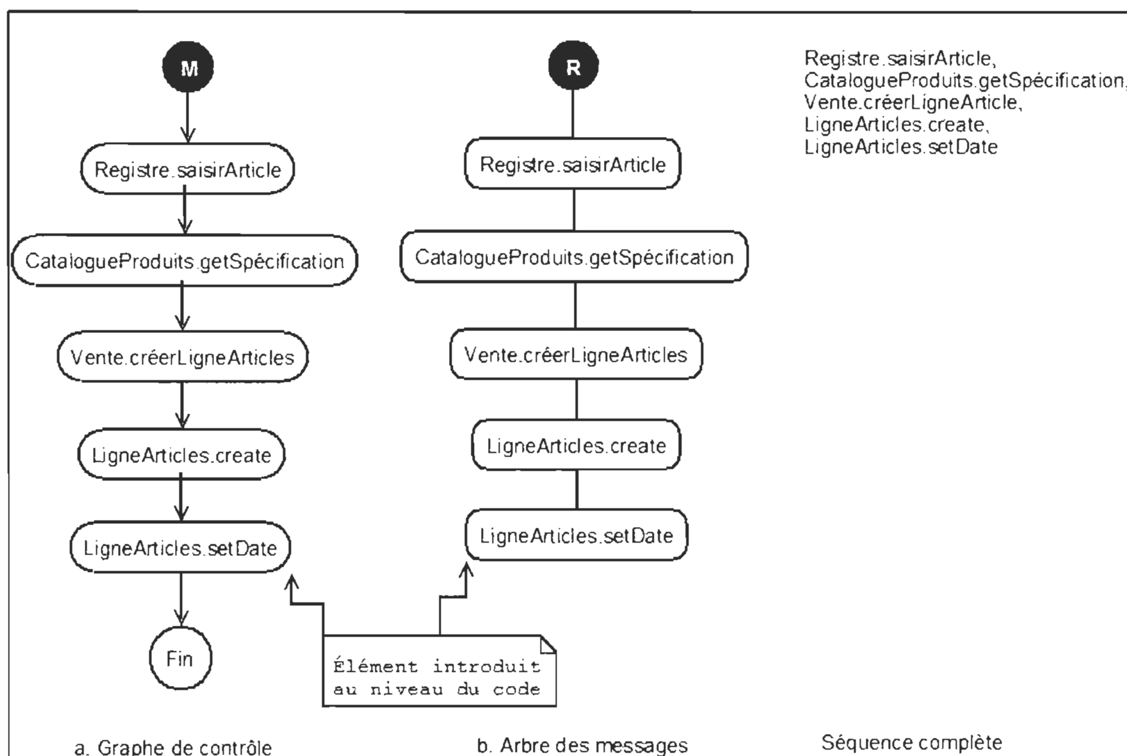


Figure 12 Génération de la séquence principale pour la collaboration 'saisirArticle'.

A partir du graphe de contrôle l'arbre des messages est généré. Il est illustré à la section 'b' de la figure 12. Dans le cas de cet exemple, la collaboration ne contient qu'un seul chemin d'exécution, c'est pourquoi il n'y a qu'une seule branche. Par la suite, il suffit de parcourir l'arbre des messages afin d'identifier les séquences réduites pour la collaboration en question. La séquence réduite est illustrée à la section droite de la figure 12 sous forme d'expression régulière. Comme il n'y a qu'un seul chemin d'exécution, il n'y a qu'une séquence. Maintenant que les séquences réduites ont été identifiées pour toutes les collaborations impactées, il faut identifier tous les cas d'utilisations qui sont à leur tour impactés par ces collaborations. Dans le cas de cet exemple, seul le cas d'utilisation 'Traiter une vente' illustré à la figure 13 est impacté par la collaboration 'saisirArticle' (figure 11).

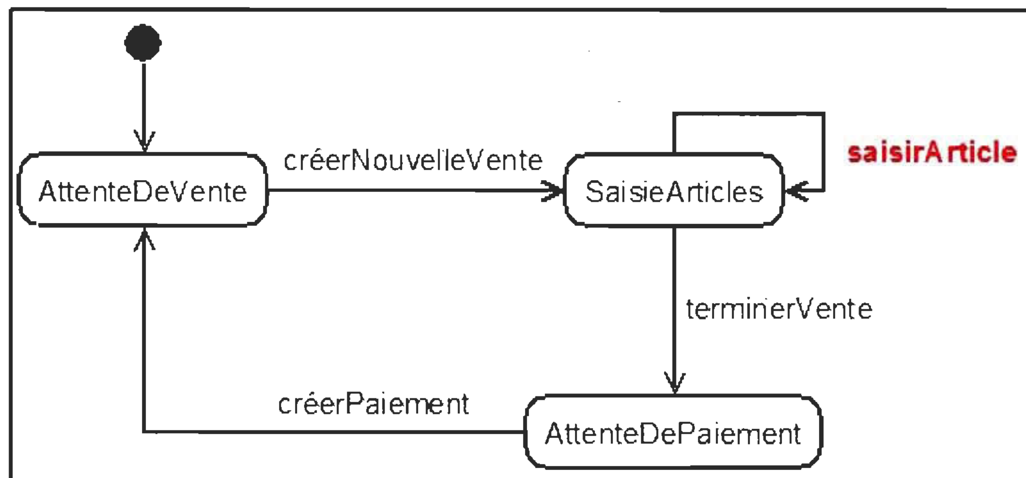


Figure 13 Diagramme d'états du cas d'utilisation 'traiter une vente'.

Maintenant, les séquences complètes de tests doivent être générées. Pour ce faire, le diagramme d'états du cas d'utilisation 'traiter une vente' est utilisé afin d'en générer le graphe de contrôle, l'arbre des messages et la séquence principale sous forme d'expression régulière. Cette étape est illustrée à la figure 14.

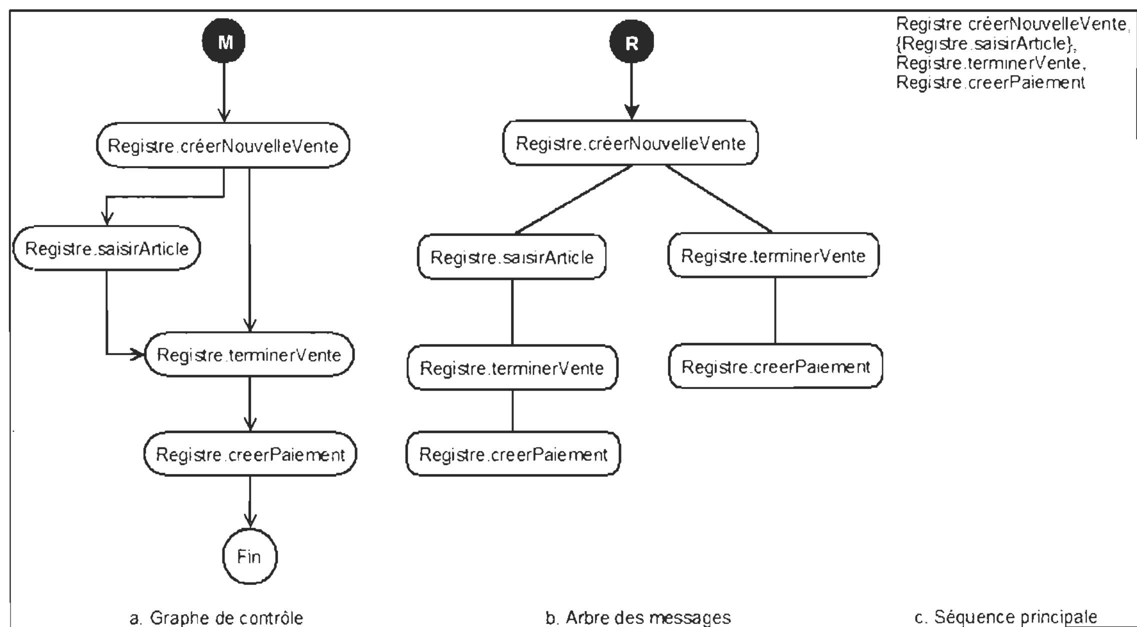


Figure 14 Génération de la séquence principale pour le cas d'utilisation.

Suite à la génération de la séquence principale, on remarque l'existence de deux séquences pour ce cas :

1. Registre.créerNouvelleVente, Registre.terminerVente, Registre.créerPaiement.
2. Registre.créerNouvelleVente, Registre.saisirArticle, Registre.terminerVente, Registre.créerPaiement.

Puisque la modification porte sur la collaboration initiée par la méthode 'saisirArticle' (figure 11), la séquence 1 ne sera pas affectée par la modification. Seule la séquence 2 est retenue. Maintenant que les séquences principales et réduites ont été générées, la dernière étape consiste en l'expansion des séquences. Il s'agit de remplacer la méthode 'saisirArticle' par sa propre séquence réduite. La séquence finale de tests est illustrée par la figure 15.

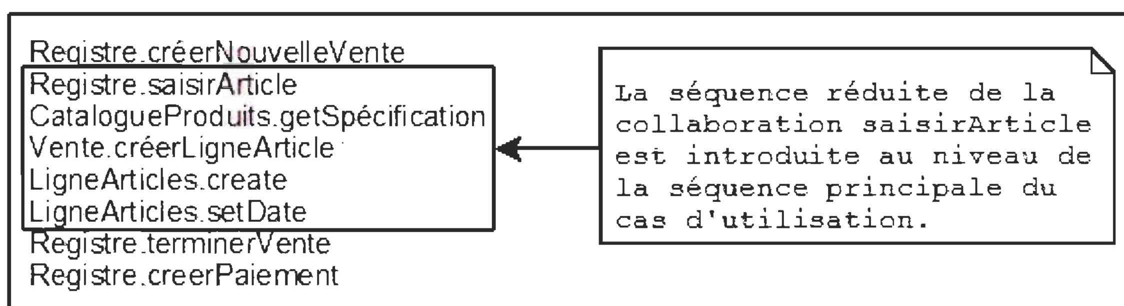


Figure 15 Séquence finale de tests pour l'exemple.

## 6.5 Génération automatique des tests

### 6.5.1 Objectif

Afin d'effectuer des gains en efficacité et diminuer l'effort requis pour effectuer les tests de régression, la génération automatique de tests est utilisée. Suite à la génération des séquences à re-tester, une classe de test est générée avec les stubs des méthodes et des séquences devant être re-testées en se basant sur l'approche de Bourque-Fortin et al. [Bourque07].

### 6.5.2 Méthode

Une fois la classe de test générée, le développeur peut inscrire les tests qu'il désire effectuer au sein des différents stubs. Par la suite, ces tests seront exécutés de façon automatisée grâce à l'outil JUnit. L'outil JUnit est une plateforme qui facilite la production et l'exécution de tests unitaires en java [JUnit01]. Cette plateforme sera utilisée afin d'automatiser l'exécution des tests. Suite à la génération des séquences de tests, une classe est créée. Cette classe hérite de la superclasse TestCase, ce qui indique l'utilisation de JUnit. Pour chaque méthode au sein des séquences, une méthode de test unitaire est créée. Par la suite, pour chaque séquence, une méthode est créée et cette dernière ne fait qu'appeler dans l'ordre les différentes méthodes de tests unitaires créées précédemment. Suite à cette génération automatique, le développeur peut compléter le corps des méthodes de tests unitaires et lancer l'exécution en sachant que ces tests couvrent l'ensemble des éléments impactés par les modifications. La figure 16 illustre un exemple d'une telle classe de test générée automatiquement. Suite à cette étape, une batterie de tests de régression est assemblée de façon automatique.

```
public class TestsSequences extends TestCase {

public static void main(String[] args) {}

public void setUp() {}

public void tearDown() {}

// Tests unitaires
public void Registre_creerNouvelleVente() {}

public void Registre_saisirArticle() {}

public void CatalogueProduits_getSpecification() {}

public void Vente_creerLigneArticles() {}

public void LigneArticles_New() {}

public void Registre_terminerVente() {}

public void Registre_creerPaiement() {}

public void Vente_creerPaiement() {}

public void Paiement_creerPaiement() {}

// Séquences (tests d'intégration)
public void testSequence1() {

    Registre_creerNouvelleVente();
    Registre_saisirArticle();
    CatalogueProduits_getSpecification();
    Vente_creerLigneArticles();
    LigneArticles_New();
    Registre_terminerVente();
    Registre_creerPaiement();
    Vente_creerPaiement();
    Paiement_creerPaiement();
}

public void testSequence2() {

    Registre_creerNouvelleVente();
    Registre_terminerVente();
    Registre_creerPaiement();
    Vente_creerPaiement();
    Paiement_creerPaiement();
}
}
```

Figure 16 Exemple de génération automatique des tests.

## 6.6 Conclusion

L'algorithme de génération des séquences de tests est présenté et un exemple concret permet d'en faciliter la compréhension. De plus, la génération automatique des cas de tests est abordée. Le chapitre 7 présente le protocole d'expérimentation ainsi que l'étude de cas utilisé pour l'expérimentation et finalement les résultats sont présentés.

# CHAPITRE 7

# EXPÉRIMENTATION

## 7.1 Protocole d'expérimentation

### 7.1.1 Objectif

L'objectif de l'expérimentation se défini en deux volets. Dans un premier temps, le fonctionnement de l'approche est démontré. Par la suite, une évaluation empirique de l'approche, basée sur plusieurs critères, est effectuée. Cette évaluation permettra d'établir les gains qu'offre l'approche adoptée.

### 7.1.2 Démarche

Afin d'atteindre les objectifs mentionnés précédemment, une étude de cas concrète est réalisée. Pour ce faire, le logiciel « NextGen » sert d'élément de base. Il l'agit d'un système de point de vente rédigé en langage Java et développé par Craig Larman [Larman01]. La portée de ce dernier est étendue au-delà des limites du système d'origine afin de mieux répondre aux besoins de notre étude. Des modifications sont simulées au niveau du logiciel résultant de sorte à générer quelques versions successives. L'ensemble de ces modifications couvre les différentes possibilités de changement de sorte à garantir un haut niveau de confiance face au fonctionnement de l'approche.

L'approche étant pilotée par les cas d'utilisation, voici les changements qui ont un intérêt :

a. ajout d'un cas d'utilisation;
    a. ajout d'un diagramme d'états-transitions;
    b. ajout d'un ou de plusieurs diagrammes de collaboration;
    c. ajout du code source (classes et/ou méthodes);

b. modification d'un cas d'utilisation;
   a. modification d'un diagramme d'états-transitions;
   b. modification d'un ou plusieurs diagrammes de collaboration;
   c. modification d'une ou plusieurs classes et/ou méthodes;

c. suppression d'un cas d'utilisation;
   a. suppression d'un diagramme d'états;
   b. suppression d'un ou de plusieurs diagrammes de collaboration;
   c. suppression d'une ou de plusieurs classes et/ou méthodes.

### 7.1.3 Critères

Afin d'évaluer les gains effectués par l'utilisation de l'approche, des critères doivent être définies. Une revue de la littérature concernant les critères d'évaluation des différentes approches proposées a été présentée et discutée au chapitre 2. Suite à cette revue, deux critères ont été sélectionnées :

a. La réduction de la suite de tests est déterminée en fonction du nombre de séquences de tests qui sont éliminées (ne seront pas re-testées) comparativement au nombre total de séquences de tests;

b. La réutilisation de la suite de tests est déterminée en fonction du taux de réutilisation au sein des séquences de test sélectionnées. Pour chaque séquence de tests sélectionnée, une comparaison est faite avec la séquence correspondante (si applicable) de la version i du logiciel. Le taux de réutilisation local est le nombre de méthodes qui faisaient partie de la séquence correspondante de la version i comparativement au nombre total de méthodes dans la séquence de la version i+1. Pour obtenir le taux de réutilisation global, il suffit de l'appliquer à l'échelle de toutes les séquences sélectionnées.

### 7.1.4 Protocole

En supposant que la version originale de l'étude de cas est décrit par la variable P, P' représente la version modifiée (successive) de P. De la même façon, P'' représente la version modifiée (successive) de P' et ainsi de suite.

Pour chaque version modifiée de P :

a. appliquer l'approche sur la version modifiée (P') et la version de départ (P);
b. recueillir les résultats;
c. analyser et interpréter les résultats en fonction des critères énoncés précédemment.

### 7.2 Étude de cas et résultats

L'étude de cas est basée sur le système de point de vente présenté par Craig Larman dans [Larman01]. Elle est constituée de neuf cas d'utilisations, donc neuf diagrammes d'états et plusieurs diagrammes de collaborations. Au niveau du code source, la programmation a été faite à l'aide du langage Java. On retrouve 7 classes et 33 méthodes. Afin de pouvoir expérimenter l'approche, quatre versions successives sont crées. Chacune de ces versions introduit des modifications à divers niveaux afin de tester les divers cas possibles. Dans un premier temps, les diagrammes d'états et de collaboration de la version de départ (V1) sont illustrés en ordre de cas d'utilisation. Par la suite, pour chaque version successive, les modifications apportées sont décrites et les modèles impliqués sont illustrés.

De plus, les résultats d'expérimentations sont présentés de façon détaillée pour chaque itération. Tel que spécifié précédemment, les deux critères servant à évaluer l'approche proposée sont :

i. La réduction de la suite de tests : combien de séquences sont affectées par les modifications;
ii. La réutilisation de la suite de tests : combien de cas de tests est-il possible de réutiliser à partir de la batterie de test de la version i.

Étant donné le nombre très peu élevé d'approches portant sur le test de régression basées sur les modèles UML, et considérant que celles qui existent se basent sur d'autres modèles que ceux utilisés par l'approche proposée, il semble hasardeux de tenter une comparaison. Par conséquent, l'approche proposée sera comparée avec les deux façons de faire suivantes :

i. Tester toutes les séquences de chaque cas d'utilisation affecté ou modifié;
ii. Tester toutes les séquences du programme au grand complet (aussi connu sous le nom de 'retest-all').

### 7.2.1 Traiter une vente

Ce cas d'utilisation permet la saisie d'une vente et des articles qui la compose par un commis. La figure 17 représente le diagramme d'états le décrivant.

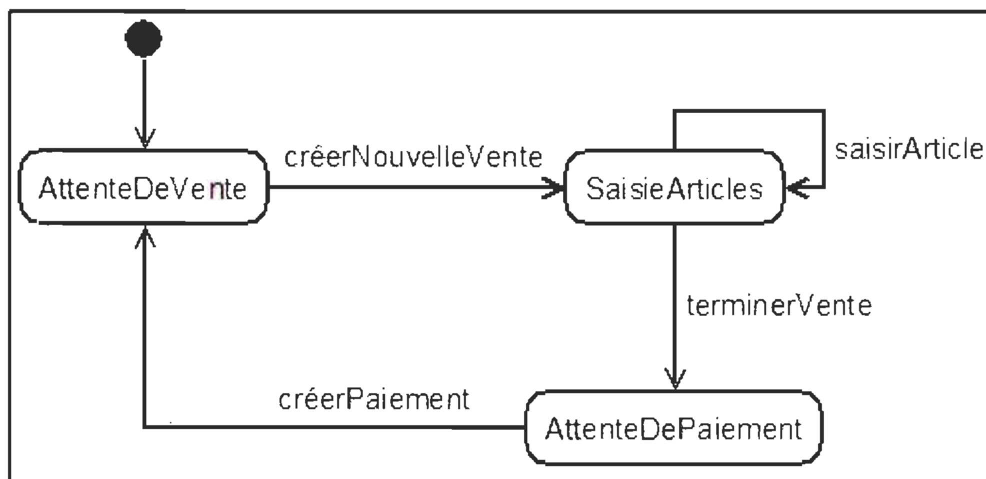


Figure 17 Diagramme d'état du cas d'utilisation 'Traiter une vente'.

Le diagramme de collaboration 'saisirArticle' présenté à la figure 18 illustre le déroulement normal de la saisie d'un article par le commis.

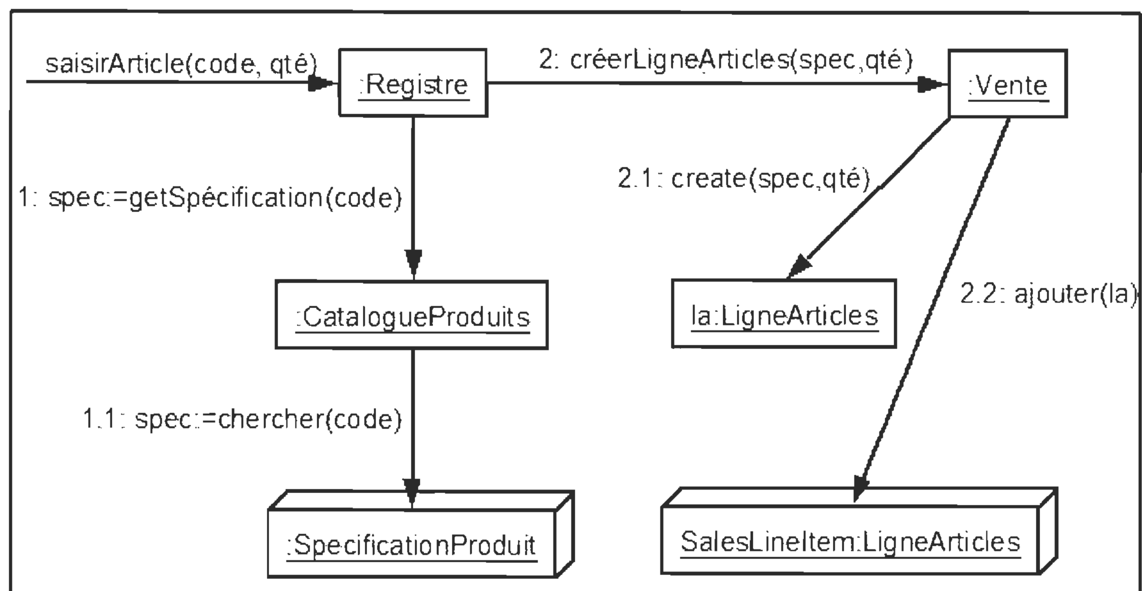


Figure 18 Diagramme de collaboration du message 'saisirArticle'.

Le diagramme de collaboration 'créerPaiement' présenté à la figure 19 permet la saisie du paiement suite à la conclusion de la vente.

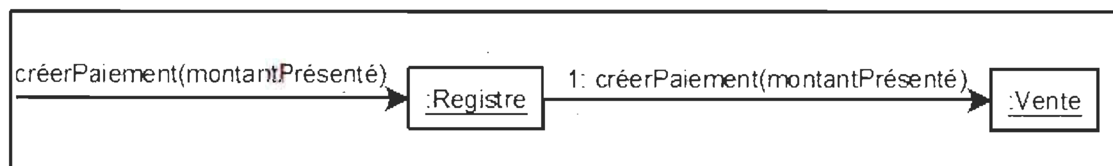


Figure 19 Diagramme de collaboration du message 'créerPaiement'.

### 7.2.2 Sécurité

Le cas d'utilisation 'Sécurité' illustre le fonctionnement du processus d'identification des utilisateurs. La figure 20 illustre le diagramme d'états le décrivant.

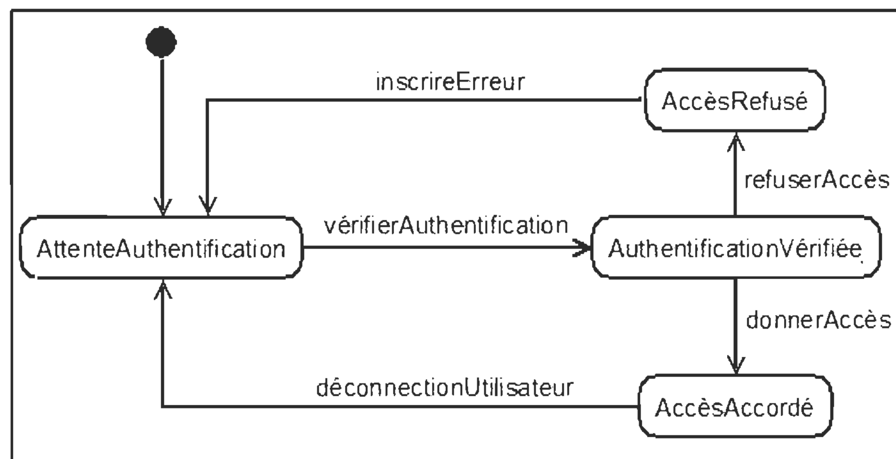


Figure 20 Diagramme d'état du cas d'utilisation 'Sécurité'.

Le diagramme de collaboration 'vérifierAuthentification' présenté à la figure 21 permet de déterminer, suite à la saisie d'un nom d'utilisateur et d'un mot de passe, si ces derniers sont valides.

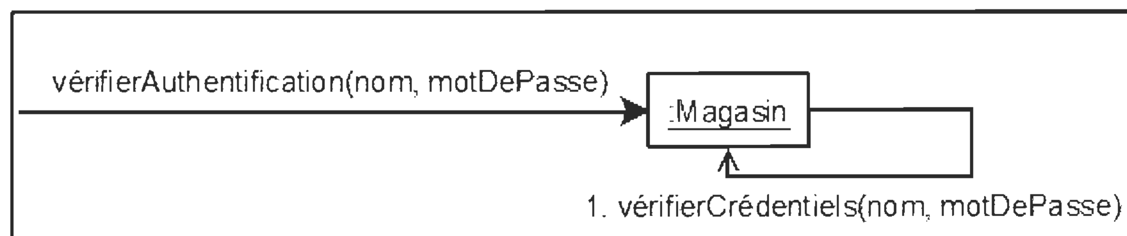


Figure 21 Diagramme de collaboration du message 'vérifierAuthentification'.

### 7.2.3 Ajout, modification et suppression d'un utilisateur

Le cas d'utilisation 'gestion des utilisateurs' permet l'ajout, la modification et la suppression des utilisateurs qui ont accès au système. La figure 22 illustre le diagramme d'états le décrivant.

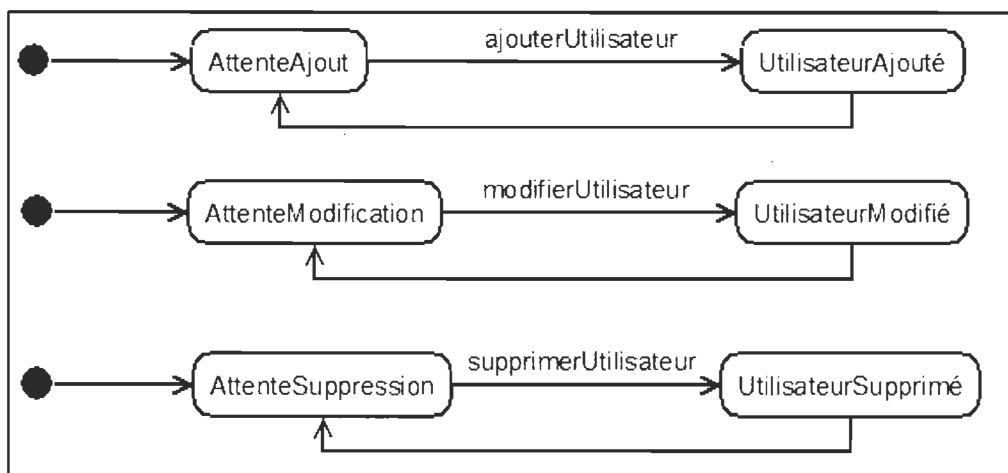


Figure 22 Diagrammes d'états des cas d'utilisation 'Ajouter un utilisateur', 'Modifier un utilisateur' et 'Supprimer un utilisateur', de haut en bas respectivement.

### 7.2.4 Analyser les ventes

Le cas d'utilisation 'Analyser les ventes' permet la journalisation des ventes lors de chaque fin de journée. La figure 23 illustre le diagramme d'états le décrivant.

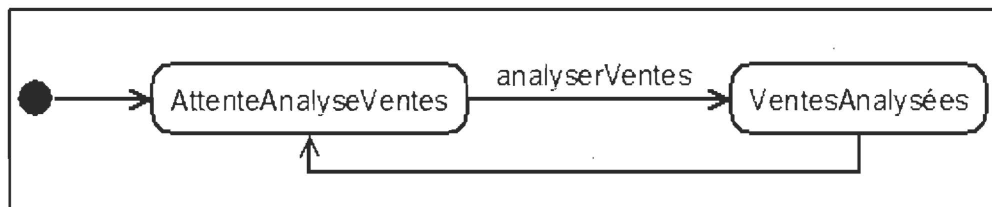


Figure 23 Diagramme d'états du cas d'utilisation 'Analyser les ventes'.

Le diagramme de collaboration de 'AnalyserVentes' présenté à la figure 24 illustre la procédure pour journaliser les ventes.

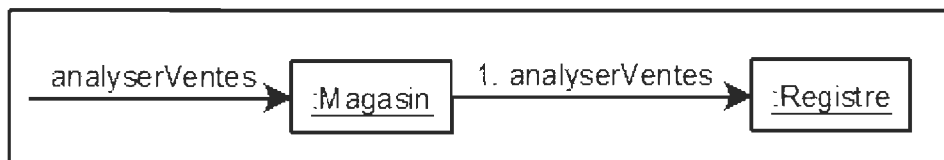


Figure 24 Diagramme de collaborations du message 'AnalyserVentes'.

### 7.2.5 Démarrer le système

Le cas d'utilisation 'Démarrer le système' illustre le déroulement normal du démarrage du système. La figure 25 illustre le diagramme d'états le décrivant.

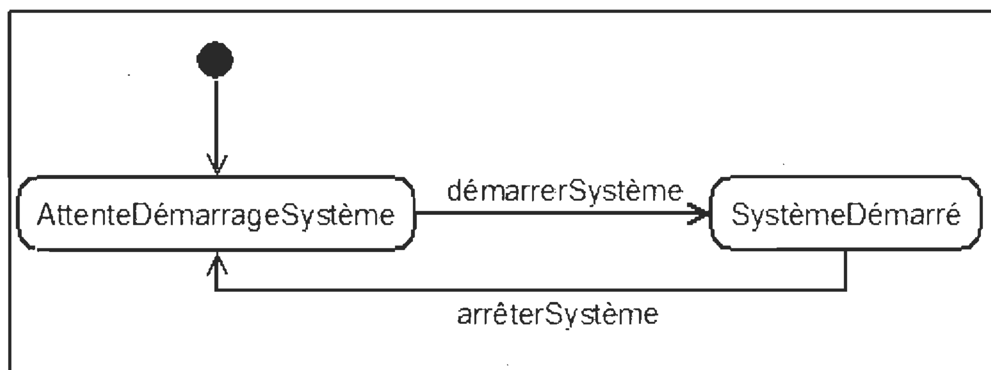


Figure 25 Diagramme d'états du cas d'utilisation 'DémarrerSystème'.

### 7.2.6 Arrêter le système

Le cas d'utilisation 'Arrêter le système' illustre le déroulement normal de l'arrêt du système. La figure 26 illustre le diagramme d'états le décrivant.

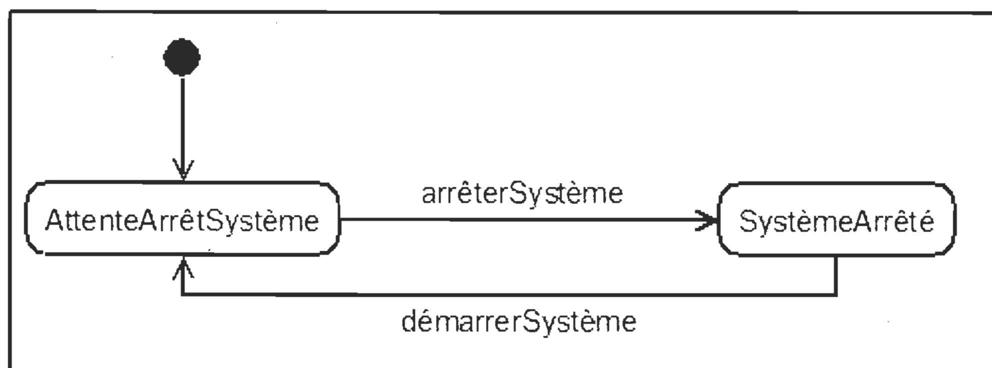


Figure 26 Diagramme d'états du cas d'utilisation 'Arrêter système'.

### 7.2.7 Gérer les tables

Le cas d'utilisation 'Gérer les tables' illustre le déroulement normal de la gestion et de la maintenance des tables du système. La figure 27 illustre le diagramme d'états le décrivant.

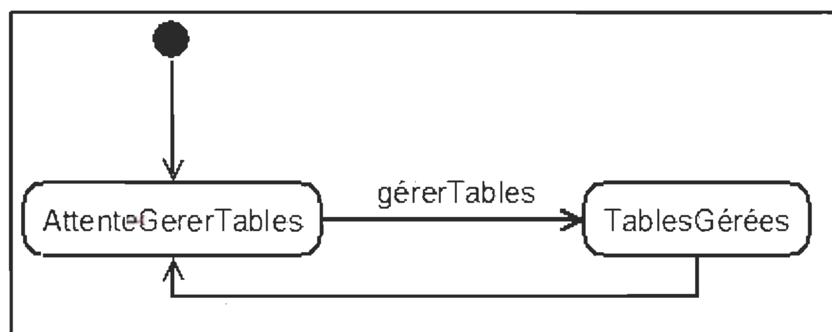


Figure 27 Diagramme d'états du cas d'utilisation 'Gérer les tables'.

### 7.2.8 Version V2

Lors de cette itération on procède à une modification profonde du cas d'utilisation 'Traiter une vente'. Le paiement par crédit va être ajouté, ce qui implique une modification du diagramme d'états du cas d'utilisation (figure 28), l'ajout d'un diagramme de collaboration (figure 29) et la modification d'une classe (introduction de deux nouvelles méthodes).

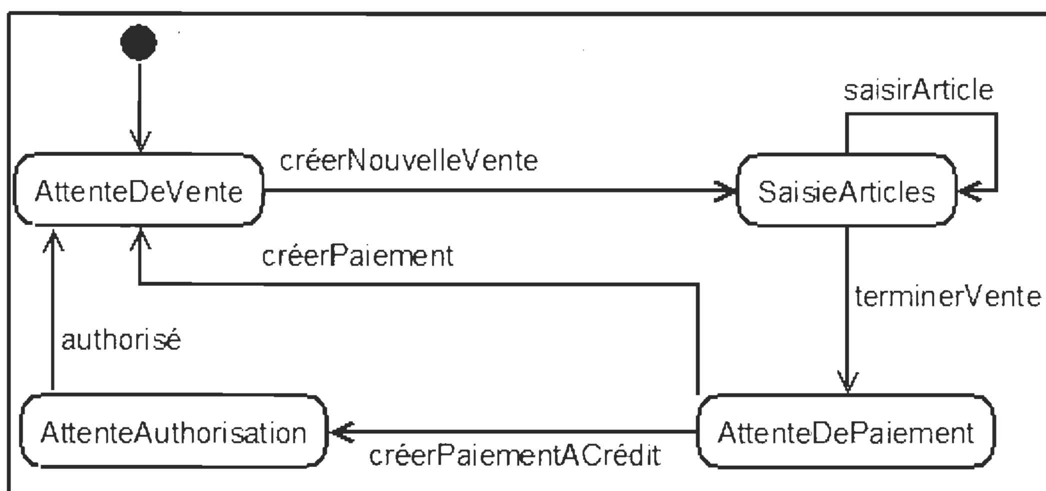


Figure 28 Diagramme d'états modifié du cas d'utilisation 'Traiter une vente'.

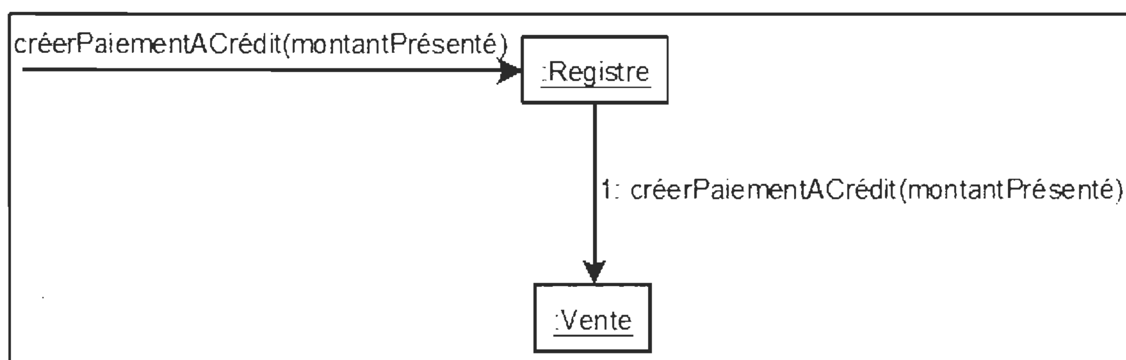


Figure 29 Diagramme de collaborations (nouveau) du message 'créerPaiementACrédit'.

Les figures 28 et 29 illustrent respectivement le diagramme d'états du cas d'utilisation et le diagramme de collaboration. Cette modification a pour conséquence directe que deux nouvelles séquences sont ajoutées. Le cas d'utilisation 'traiter une vente' en compte dorénavant quatre. Selon l'approche proposée, seulement deux de ces quatre séquences doivent être testées.

#### 7.2.8.1 Résultats de l'itération 1 (Passage de la version V1 à V2)

Séquences sélectionnées :

```
SEQUENCES DE TESTS SÉLECTIONNÉES:

1: NextGen.Registre.creerNouvelleVente, NextGen.Registre.saisirArticle,
NextGen.Registre.terminerVente, NextGen.Registre.creerPaiementCredit

2: NextGen.Registre.creerNouvelleVente, NextGen.Registre.terminerVente,
NextGen.Registre.creerPaiementCredit
```

Critère 1 : Réduction de la suite de tests
Nombre de séquences sélectionnées: 2
Nombre total de séquences qui composent les cas d'utilisations affectés : 4
Nombre total de séquences qui composent le programme : 13

Gain par rapport au retest par cas d'utilisation : 50%
Gain par rapport au retest-all : 84.61%

Critère 2 : Réutilisation de la suite de tests
Nombre de cas de test réutilisables: 5
Nombre de cas de test total: 7
Taux de réutilisation : 71.43%

Autres indicateurs
Nombre de méthodes sélectionnées: 4
Nombre de méthodes totales: 31

### 7.2.9 Version V3

Lors de cette itération, le diagramme de collaboration 'créerPaimentÀCrédit' du cas 'traiter une vente' subit un changement et un nouveau diagramme de collaboration est crée pour 'refuserAccès' du cas d'utilisation 'sécurité'. Ces changements ont comme impact qu'au niveau du cas 'traiter une vente', deux séquences sur un total de quatre doivent être re-testées. Au niveau du cas 'sécurité', une séquence sur un total de deux doit être re-testée. Au total, trois séquences seront sélectionnées. La figure 30 illustre le diagramme de

collaboration modifié de 'créerPaiementÀCrédit'. La figure 31 illustre le nouveau diagramme de collaboration de 'refuserAccès'.

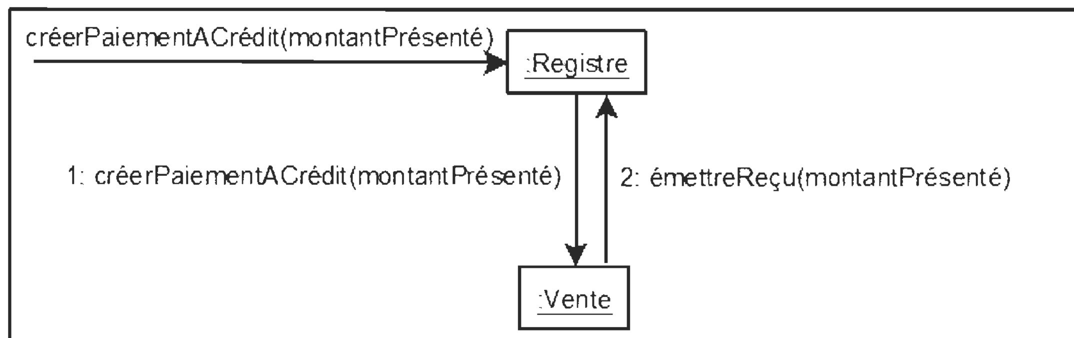


Figure 30 Diagramme de collaboration modifié de 'créerPaiementÀCrédit'.

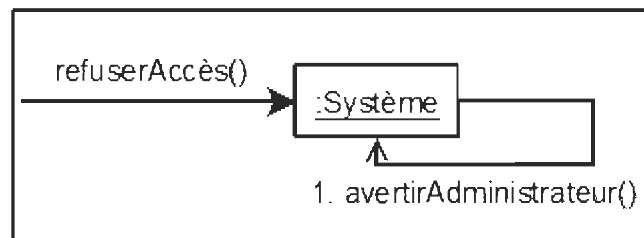


Figure 31 Diagramme de collaboration de 'refuserAccès'.

#### 7.2.9.1 Résultats de l'itération 2 (Passage de la version V2 à V3)

Séquences sélectionnées :

| SEQUENCES DE TESTS SÉLECTIONNÉES:<br><br>1: NextGen.Systeme.Authentification, NextGen.Systeme.refuserAcces(),<br>NextGen.Systeme.avertirAdministrateur(), NextGen.Systeme.inscrireErreur() |
|---|

2: NextGen.Registre.creerNouvelleVente, NextGen.Registre.saisirArticle, NextGen.Registre.terminerVente, NextGen.Registre.creerPaiementCredit(), NextGen.Vente.creerPaiementCredit(), NextGen.Registre.emettreRecu()

3: NextGen.Registre.creerNouvelleVente, NextGen.Registre.terminerVente, NextGen.Registre.creerPaiementCredit(), NextGen.Vente.creerPaiementCredit(), NextGen.Registre.emettreRecu()

Critère 1 : Réduction de la suite de tests
Nombre de séquences sélectionnées: 3
Nombre total de séquences qui composent les cas d'utilisations affectés : 6
Nombre total de séquences qui composent le programme : 15

Gain par rapport au retest par cas d'utilisation : 50%
Gain par rapport au retest-all : 80%

Critère 2 : Réutilisation de la suite de tests
Nombre de cas de test réutilisables: 11
Nombre de cas de test total: 17
Taux de réutilisation : 64.71%

Autres indicateurs
Nombre de méthodes sélectionnées: 10
Nombre de méthodes totales: 33

### 7.2.10 Version V4

Lors de cette itération, la signature de la méthode 'saisirArticle' est modifiée. Cela a comme conséquence que le diagramme d'états est modifié et il en est de même pour le diagramme de collaboration de la méthode 'saisirArticle'. La figure 32 illustre le diagramme d'états modifié et la figure 33 illustre le diagramme de collaboration modifié. Suite à cette modification, deux séquences sont sélectionnées.

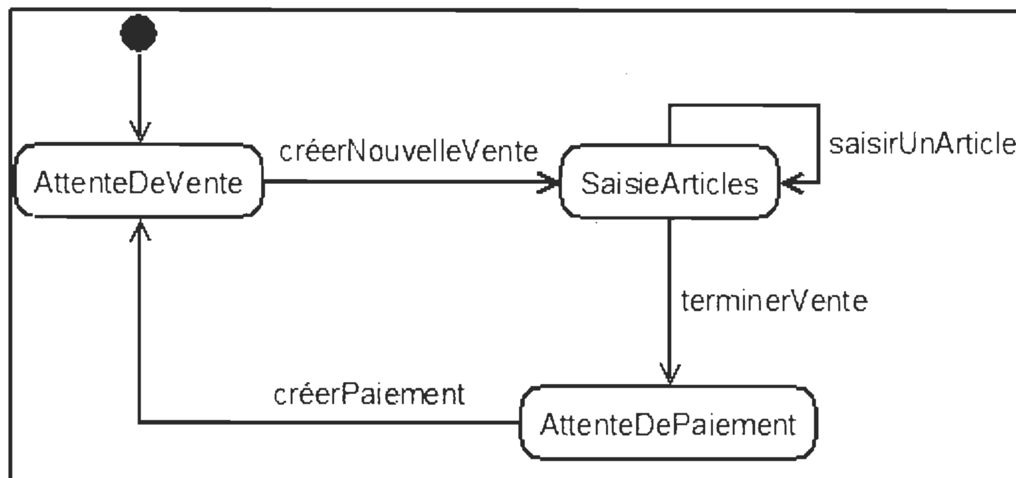


Figure 32 Diagramme d'états modifié du cas d'utilisation 'traiter une vente'.

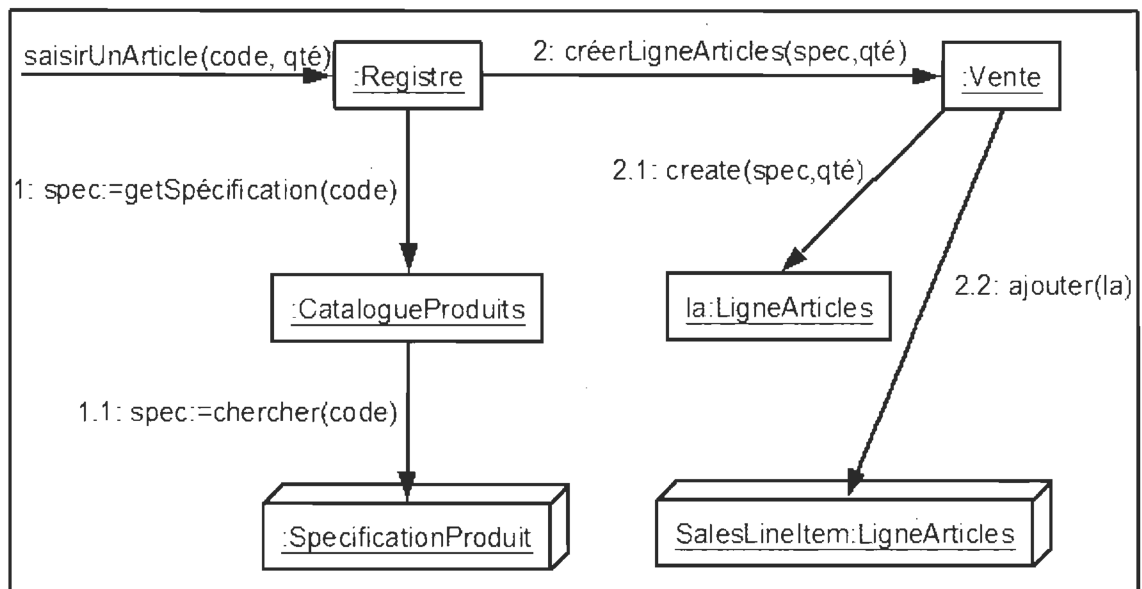


Figure 33 Diagramme de collaboration modifié de la méthode 'saisirUnArticle', auparavant nommée 'saisirArticle'.

#### 7.2.10.1 Résultats de l'itération 3 (Passage de la version V3 à V4)

Séquences sélectionnées :

| SEQUENCES DE TESTS SÉLECTIONNÉES: |
|---|
| 1:NextGen.Registre.creerNouvelleVente, NextGen.Registre.saisirUnArticle, NextGen.CatalogueProduits.getSpecification, NextGen.Vente.creerLigneArticles, LigneArticles.New, NextGen.List<LigneArticles>.add, NextGen.LigneArticles.New, NextGen.Registre.terminerVente, NextGen.Registre.creerPaiement |
| 2: NextGen.Registre.creerNouvelleVente, NextGen.Registre.saisirUnArticle, NextGen.CatalogueProduits.getSpecification, NextGen.Vente.creerLigneArticles, LigneArticles.New, NextGen.List<LigneArticles>.add, NextGen.LigneArticles.New, NextGen.Registre.terminerVente, NextGen.Registre.creerPaiementCredit |

<u>Critère 1 : Réduction de la suite de tests</u>
Nombre de séquences sélectionnées: 2
Nombre total de séquences qui composent les cas d'utilisations affectés : 4
Nombre total de séquences qui composent le programme : 15
Gain par rapport au retest par cas d'utilisation : 50%
Gain par rapport au retest-all : 86.67%

<u>Critère 2 : Réutilisation de la suite de tests</u>
Nombre de cas de test réutilisables: 12
Nombre de cas de test total: 18
Taux de réutilisation : 66.67%

Autres indicateurs
Nombre de méthodes sélectionnées: 10
Nombre de méthodes totales: 33

### 7.2.11 Version V5

Lors de cette itération, plusieurs changements sont apportés au niveau du code (modifications de bas niveau) mais pas au niveau des modèles. Au total, les trois méthodes suivantes sont modifiées : saisirArticle, refuserAccès et analyserVentes. Ces modifications entraînent la sélection de quatre séquences de tests.

### 7.2.11.1 Résultats de l'itération 4 (Passage de la version V4 à V5)

Séquences sélectionnées :

SEQUENCES DE TESTS SÉLECTIONNÉES:

1: NextGen.Magasin.analyserVentes

2: NextGen.Systeme.Authentification, NextGen.Systeme.refuserAcces, NextGen.Systeme.avertirAdministrateur, NextGen.Systeme.inscrireErreur

3: NextGen.Registre.creerNouvelleVente, NextGen.Registre.saisirUnArticle, NextGen.CatalogueProduits.getSpecification, NextGen.Vente.creerLigneArticles, LigneArticles.New, List<LigneArticles>.add, NextGen.LigneArticles.New, NextGen.Registre.terminerVente, NextGen.Registre.creerPaiement

4: NextGen.Registre.creerNouvelleVente, NextGen.Registre.saisirUnArticle, NextGen.CatalogueProduits.getSpecification, NextGen.Vente.creerLigneArticles, LigneArticles.New, List<LigneArticles>.add, NextGen.LigneArticles.New, NextGen.Registre.terminerVente, NextGen.Registre.creerPaiementCredit

Critère 1 : Réduction de la suite de tests
Nombre de séquences sélectionnées: 4
Nombre total de séquences qui composent les cas d'utilisations affectés : 7
Nombre total de séquences qui composent le programme : 15

Gain par rapport au retest par cas d'utilisation : 42.85%
Gain par rapport au retest-all : 73.33%

Critère 2 : Réutilisation de la suite de tests
Nombre de cas de test réutilisables: 17
Nombre de cas de test total: 23
Taux de réutilisation : 73.91%

Autres indicateurs
Nombre de méthodes sélectionnées: 15
Nombre de méthodes totales: 33

### 7.2.12 Illustration des résultats

Le premier graphique (figure 34) présente une comparaison entre l'approche proposée (en bleu) ainsi que les deux approches suivantes:

a. (En rouge) Retester toutes les séquences de tous les cas d'utilisation affectés ou modifiés;
b. (En vert) Retester toutes les séquences pour le programme au grand complet.

Pour chaque itération, on y retrouve le nombre de séquences sélectionnées pour chaque approche.

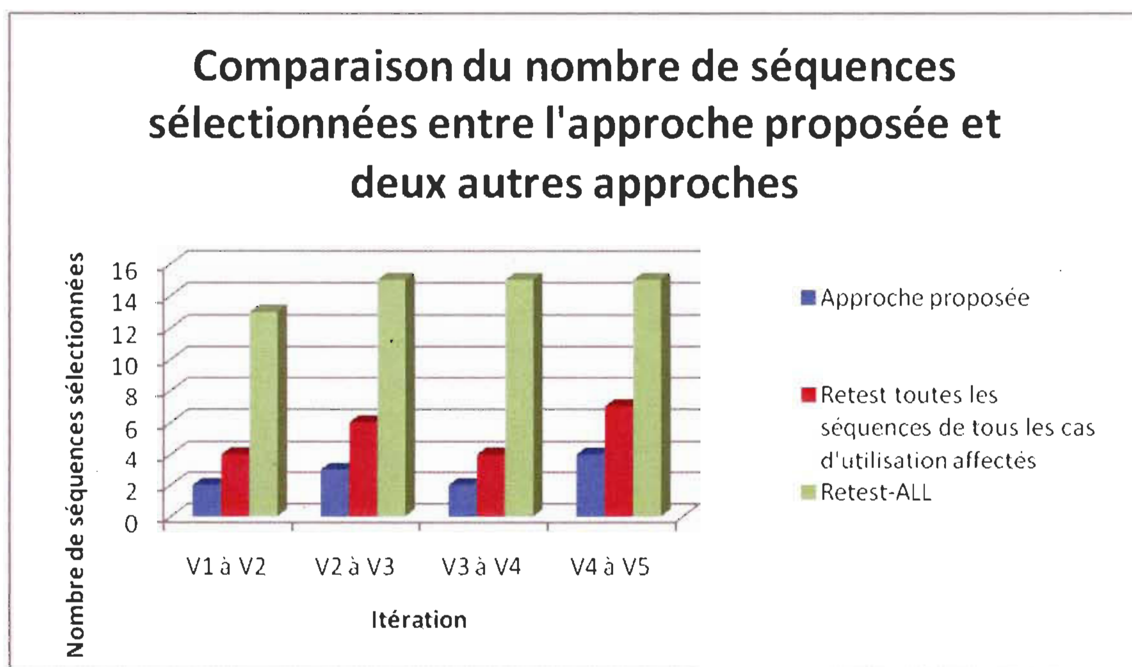


Figure 34 Comparaison du nombre de séquences sélectionnées entre l'approche proposée et deux autres approches.

Le second graphique (figure 35) représente le taux de réutilisation des cas de test pour l'approche proposée lors de chacune des itérations.

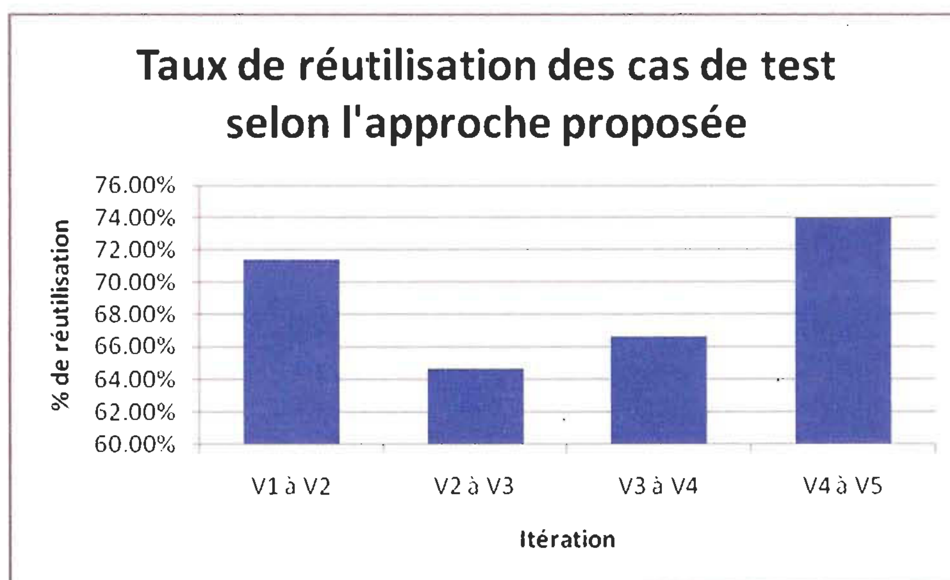


Figure 35 Taux de réutilisation des cas de test selon l'approche proposée lors des expérimentations.

### 7.3 Conclusion

Le protocole d'expérimentation est défini et l'étude de cas est décrite en détail. Les résultats sont ensuite présentés. La figure 34 illustre des gains intéressants au niveau de la réduction de la suite de tests et la figure 35 quant à elle illustre que l'approche permet la réutilisation non négligeable de plusieurs cas de tests. Le chapitre 8 décrit les différentes composantes de l'outil développé pour automatiser l'approche.

# CHAPITRE 8

## PRÉSENTATION DE L'OUTIL

Ce chapitre a pour objectif de présenter l'outil que nous avons développé pour supporter l'approche proposée. L'outil est composé de plusieurs modules. La figure 36 en illustre l'architecture générale. Étant un plugin Éclipse, l'outil est embarqué dans la plateforme Éclipse.

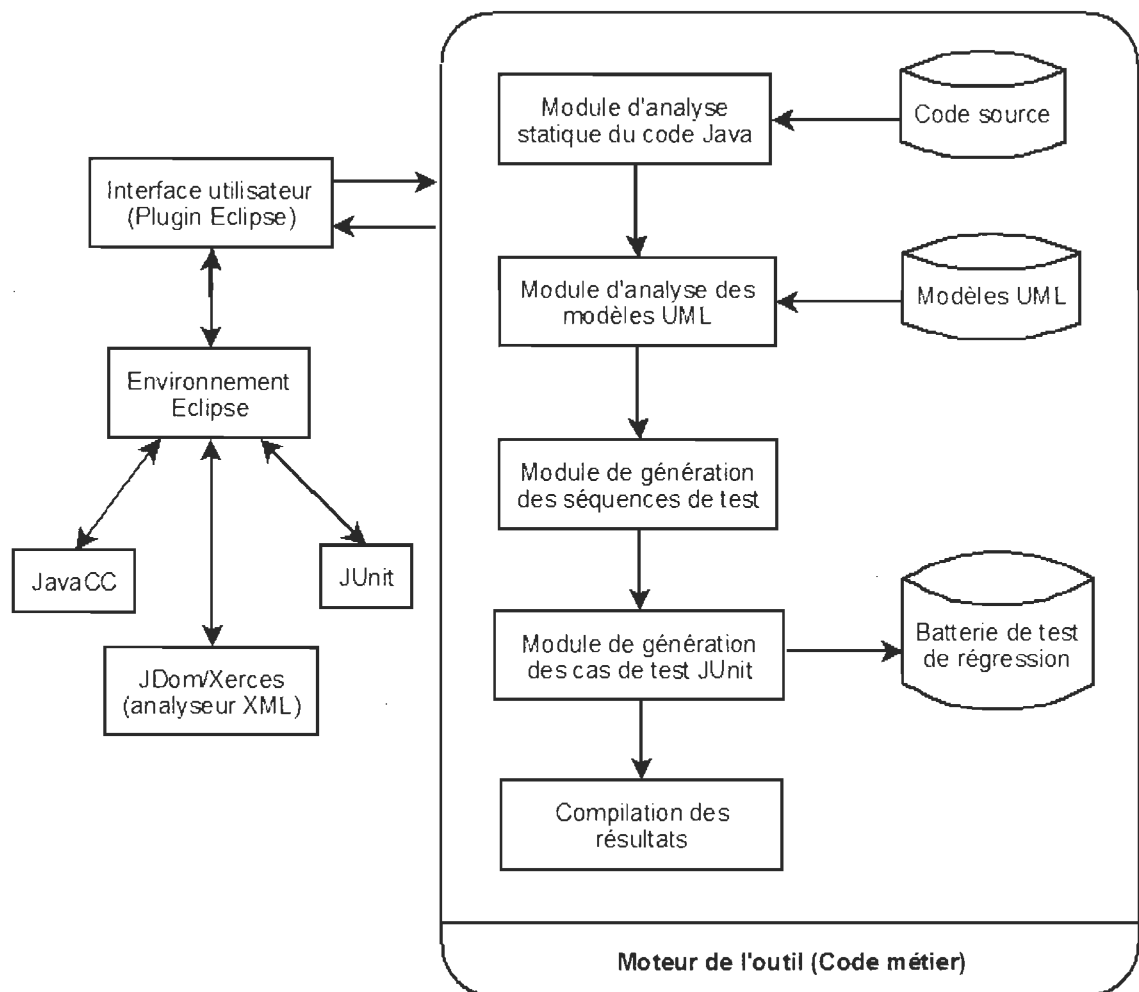


Figure 36 Architecture générale de l'outil.

## 8.1 Module d'analyse statique du code

Le module d'analyse statique du code a pour rôle de lire et d'analyser les fichiers contenant le code source Java afin de pouvoir déterminer la liste M des méthodes modifiées entre la version P et P'. Pour ce qui est de l'analyse statique comme telle, l'outil JavaCC et son complément JJTree sont utilisés. JavaCC permet, en spécifiant une grammaire, de générer un analyseur syntaxique qui transforme le code source analysé en un arbre syntaxique. Par la suite, l'utilitaire JJTree permet d'éliminer les nœuds et les branches qui n'affectent pas la sémantique du programme et ainsi l'arbre syntaxique abstrait est généré. L'arbre syntaxique abstrait est utilisé par l'outil pour déterminer si deux méthodes sont équivalentes. La figure 37 illustre le fonctionnement général du module.

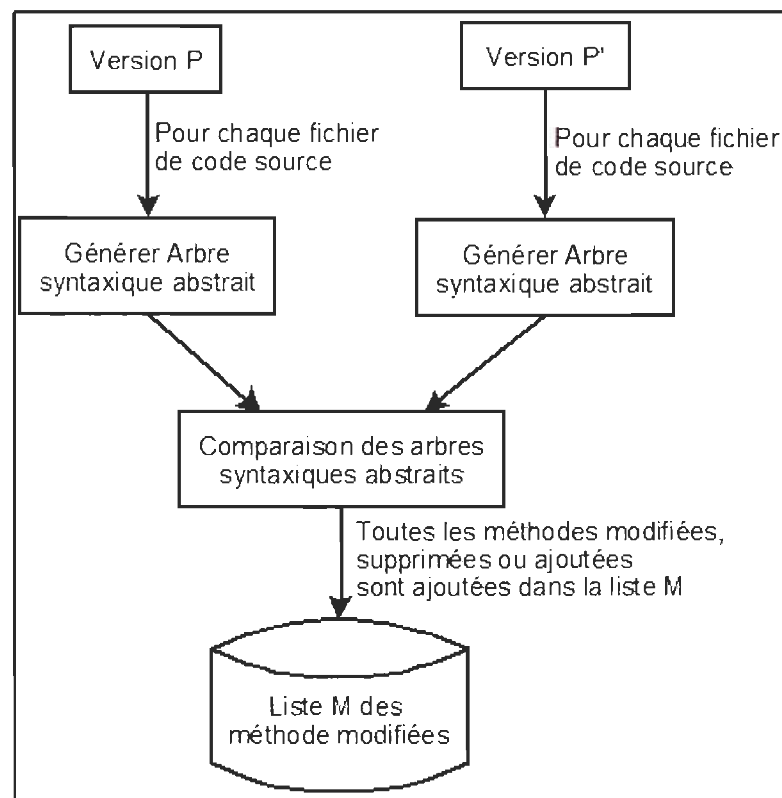


Figure 37 Fonctionnement général du module d'analyse statique.

### 8.2 Module d'analyse de modèles UML

Dans le cadre de l'approche, les modèles UML sont décrits formellement à l'aide du langage XML. XML est un langage à balise extensible qui permet à l'utilisateur de spécifier ses propres balises. La première étape consiste à créer une grammaire pour les diagrammes d'états de cas d'utilisation et une autre pour les diagrammes de collaborations. La table IV décrit la grammaire pour les diagrammes d'états de cas d'utilisation et la table V décrit la grammaire pour les diagrammes de collaborations. Afin de faciliter la saisie des divers diagrammes, un utilitaire a été créé. Cet utilitaire consiste en une interface utilisateur qui permet la déclaration des divers éléments (états, transitions, méthodes, etc.) et génère automatiquement le fichier XML final. Les figures 38 et 39 illustrent l'interface utilisateur de cet utilitaire. Enfin, les figures 40 et 41 donnent des exemples de fichiers XML.

| Balise | Description |
|---|---|
| <StatesDiagram> | Indique la déclaration d'un nouveau diagramme d'état |
| < name> | Le nom du diagramme |
| <methodesDescription> | Indique le début de la liste des méthodes |
| <methodeDescription> | Indique la déclaration d'une méthode |
| <name> | Le nom de la méthode |
| <type> | Le type de la méthode |
| <modificateur> | Le modificateur (public/privé) |
| <params> | Indique une liste de paramètres |
| <param> | Déclaration d'un paramètre |
| <name> | Nom du paramètre |
| <type> | Type du paramètre |
| <OrthogonalRegion> | Déclaration d'une région orthogonale |
| <Tag> | Nom de la région |
| <startstate> | État de départ |
| <States> | Liste d'états |
| <State> | Déclaration d'un état |
| <name> | Nom de l'état |
| <Transitions> | Liste de transitions |
| <Transtion> | Déclaration d'une transition |
| <tag> | Nom de la transition |
| <source> | État source |
| <destination> | État destination |
| <methodeDescription> | Méthode qui correspond à la transition |

Table IV Grammaire XML pour les diagrammes.

| Balise | Description |
| --- | --- |
| <Collaboration_Diagram> | Déclaration d'un diagramme de collaboration |
| <name> | Nom du diagramme |
| <object_list> | Indique une liste d'objets |
| <object> | Déclaration d'un objet |
| <name> | Nom de l'objet |
| <message_list> | Liste de messages |
| <message> | Déclaration d'un message |
| <name> | Nom du message |
| <type> | Type du message |
| <source> | Objet source |
| <destination> | Objet destination |
| <sequence> | Numéro de séquence |
| <parameter_list> | Liste de paramètres |
| <parameter> | Déclaration d'un paramètre |
| <type> | Type du paramètre |
| <name> | Nom du paramètre |

Table V Grammaire XML pour les diagrammes de collaboration.

Figure 38 Capture d'écran de l'utilitaire qui permet de saisir les éléments d'un diagramme d'état de cas d'utilisation et de générer automatiquement le fichier XML correspondant.

Figure 39 Capture d'écran de l'utilitaire qui permet de saisir les éléments d'un diagramme de collaboration et de générer automatiquement le fichier XML correspondant.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<StatesDiagram name="ExempleDiagrammeEtat">
  <methodesDescriptions>
    <methodeDescription name="AllerEtat2" type="void">
      <modificateur name="public" />
    </methodeDescription>
  </methodesDescriptions>
  <OrthogonalRegion tag="ExampleOnlyOrthogonalRegion" startState="startState">
    <States>
      <State name="startState" />
      <State name="etat2" />
    </States>
    <Transitions>
      <Transition tag="allerVersEtat2" source="startState" destination="etat2" methodeDescription="AllerEtat2" />
    </Transitions>
  </OrthogonalRegion>
</StatesDiagram>
```

Figure 40 Exemple concret de fichier XML représentant un diagramme d'états de cas d'utilisation (L'exemple correspond aux données saisies tel qu'illustré à la figure 36).

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<collaboration_diagram>
  <name>TestDiagrammeEtat</name>
  <object_list>
    <object>
      <name>Objet1</name>
      <message_list>
        <message>
          <name>toObjet1</name>
          <type>void</type>
          <source>@start</source>
          <destination>Objet1</destination>
          <sequence>1</sequence>
        </message>
      </message_list>
    </object>
    <object>
      <name>Objet2</name>
      <message_list>
        <message>
          <name>toObjet2</name>
          <type>void</type>
          <source>Objet1</source>
          <destination>Objet2</destination>
          <sequence>2</sequence>
        </message>
      </message_list>
    </object>
  </object_list>
</collaboration_diagram>
```

Figure 41 Exemple concret de fichier XML représentant un diagramme de collaboration (L'exemple correspond aux données saisies tel qu'illustré à la figure 37).

### 8.3 Fonctionnement de l'application

Le fonctionnement de l'application est très simple. Il suffit d'ouvrir la plateforme Éclipse et d'exécuter le plugin. Il suffit, par la suite, d'entrer les informations concernant les répertoires du code source, des diagrammes d'états, des diagrammes de collaboration et indiquer un répertoire où sera écrit le fichier de test. Ensuite, il s'agit simplement de cliquer sur 'Obtenir la liste des méthodes modifiées' et le traitement s'effectue en arrière plan.

Lorsque terminés, les tests sont écrits dans le répertoire indiqué et les résultats sont inscrits au niveau de la console d'Eclipse. La figure 42 illustre l'application. La figure 43 illustre un exemple de résultat.

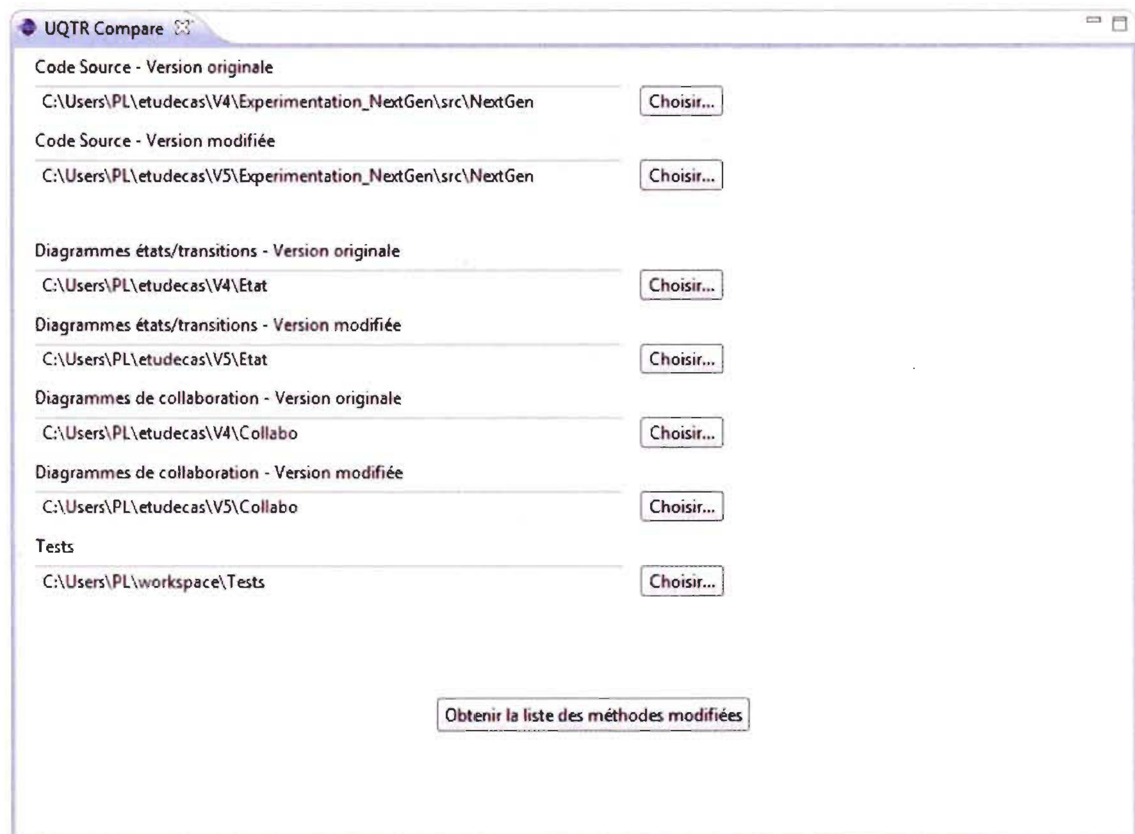


Figure 42 Interface utilisateur de l'application.

```
Problems  @ Javadoc  Declaration  Console  Properties
Eclipse Application [Eclipse Application] C:\Program Files\Java\jre6\bin\javaw.exe (2009-02-05 1:45:25 PM)
NextGen.Registre.terminerVente
NextGen.Vente.seTerminer
%%%%%%%%%%%%%%%%%%%% DEBUG %%%%%%%%%%%%%%%%%%%%%
Etat: arreterSysteme - Collab: stopSysteme
TAGGED BECAUSE OF: stopSysteme
Etat: demarrerSysteme - Collab: startSysteme
TAGGED BECAUSE OF: startSysteme
Etat: gererTables - Collab: gererTables
Etat: securite - Collab: authentification
Etat: Traiter un retour - Collab: creerNouveauRetour
Etat: Traiter un retour - Collab: saisirArticlesRetour
Etat: Traiter un retour - Collab: terminerRetour
Etat: Traiter un retour - Collab: creerRemboursement
Etat: Traiter un retour - Collab: creerRemboursementCredit
Etat: Traiter une vente - Collab: creerNouvelleVente
Etat: Traiter une vente - Collab: saisirArticles
TAGGED BECAUSE OF: saisirArticles
Etat: Traiter une vente - Collab: terminerVente
Etat: Traiter une vente - Collab: creerPaiementCredit
SEQUENCE DE TEST FINALE: NextGen.Systeme.Stop(), NextGen.Systeme.deconnecterUtilisateurs(),
SEQUENCE DE TEST FINALE: NextGen.Systeme.New, NextGen.Systeme.Start(),
SEQUENCE DE TEST FINALE: NextGen.Registre.creerNouvelleVente, NextGen.Registre.saisirArticle(), NextGen.Cata
SEQUENCE DE TEST FINALE: NextGen.Registre.creerNouvelleVente, NextGen.Registre.saisirArticle(), NextGen.Cata
Methodes uniques: NextGen.Systeme.Stop , NextGen.Systeme.deconnecterUtilisateurs , NextGen.Systeme.New , Next
Métrique 1 : Réduction de la suite de test
Nombre de séquences sélectionnées: 4
Nombre total de séquences qui composent le programme: 16
Couverture (Séquences): 25.00%

Métrique 2 : Réutilisation de la suite de test
Nombre de cas de test réutilisables: 17
Nombre de cas de test total: 22
Réutilisation de la batterie de test: 77.27%

Autres indicateurs
Nombre de méthodes sélectionnées: 14
Nombre de méthodes totales: 40
Couverture (Méthodes): 35.00%
```

Figure 43 Exemple de résultat.

## 8.4 Conclusion

L'outil développé afin d'automatiser l'approche est présenté. Les différentes composantes sont décrites et des captures d'écran sont illustrées. Le prochain chapitre présente les conclusions.

# CONCLUSIONS

Les systèmes orientés objet développés de nos jours sont de plus en plus complexes. Ces derniers sont appelés à évoluer au cours de leur cycle de vie. Les classes qui composent ces systèmes entretiennent des dépendances diverses. Ces dépendances compliquent le processus de test. Il est très difficile, voire même impossible, d'évaluer a priori l'impact d'un changement vis-à-vis du reste du système. Le test de régression permet, suite à un changement, de déterminer quelles parties du système doivent être re-testées et de supporter leur test, afin de s'assurer qu'une erreur n'ait pas été introduite par inadvertance.

Les différents travaux effectués dans le domaine des tests de régression se sont intéressés pour la plupart au code source des programmes. Il n'y a que très peu d'approches qui se sont basées sur les modèles. De plus, rares sont les approches qui automatisent le test de régression complètement.

La stratégie proposée dans ce mémoire est basée sur les modèles, particulièrement les cas d'utilisation. Elle permet d'identifier tous les cas d'utilisation qui sont affectés par une ou plusieurs modifications. La démarche adoptée permet, grâce à l'analyse des diagrammes UML qui modélisent l'aspect dynamique des cas d'utilisation, de déterminer les séquences (chemins d'exécutions) qui comportent au moins un élément modifié. Avec ces séquences, il est ensuite possible de générer une batterie de tests de régression qui permettra d'effectuer les tests tant au niveau unitaire qu'au niveau de l'intégration. Le bénéfice le plus important à l'utilisation des modèles UML est sans contredit l'indépendance à tout langage de programmation.

Une étude de cas a été réalisée en conformité avec le protocole d'expérimentation défini afin de pouvoir évaluer l'approche. Cette dernière est basée sur le système de point de vente de Craig Larman [Larman01] et adapté à nos besoins. Des changements ont été simulés, ce qui nous a permis d'avoir au total cinq versions successives de l'application. Pour chaque version P de l'application, l'approche a été appliquée sur sa version modifiée P'. L'approche

a été appliquée à toutes les versions modifiées. Les résultats de l'application de l'approche ont été recueillis au fur et à mesure. Afin d'évaluer l'approche, des critères ont été définis. Il s'agit de la réduction de la suite de tests et la réutilisation de la suite de tests. Ces critères ont été définis suite à une revue de la littérature relative aux différents critères d'évaluation utilisés par les différentes techniques de test de régression.

Les résultats obtenus suite à l'expérimentation de notre approche sur l'étude de cas démontrent que des gains tant au niveau de la réduction que de la réutilisation de la suite de tests sont effectués.

Un outil a été développé afin de supporter complètement l'approche. Ce dernier a été implémenté sous forme de plugin pour la plateforme Éclipse. Cet outil permet d'automatiser toutes les étapes décrites dans ce mémoire. Il ne suffit que de saisir les différents répertoires où se trouvent le code source de l'application, ses modèles UML et d'indiquer dans quel répertoire la batterie de test doit être créée. Un outil complémentaire a aussi été crée de sorte qu'il soit plus simple de créer des diagrammes UML au format XML grâce à une interface visuelle.

Étant donné que l'approche est basée sur les modèles UML, elle est indépendante de tout langage de programmation. De plus, il s'agit d'une approche automatisée: il ne s'agit que d'indiquer à l'outil où se trouvent les modèles et où se trouvent les deux versions à comparer. Le fait que l'outil soit intégré à l'environnement de développement Éclipse est aussi très intéressant.

## BIBLIOGRAPHIE

[Abdullah98] Abdullah, K. (1998), 'The Firewall Concept For Regression Testing and Impact Analysis of Object Oriented Systems', PhD thesis, Case Western Reserve University.

[Agrawal93] Agrawal, H.; Horgan, J. R.; Krauser, E. W. & London, S. A. (1993), Incremental regression testing, *in* 'Proc. Conference on Software Maintenance ,1993 CSM-93', pp. 348--357.

[Badri05] Badri, L.; Badri, M. & St-Yves, D. (2005), Supporting predictive change impact analysis: a control call graph based technique, *in* 'Proc. 12th Asia-Pacific Software Engineering Conference APSEC '05', pp. 167--175.

[Ball98] Ball, T. (1998), 'On The Limit of Control Flow Analysis for Regression Test Selection', in 'Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis'

[Bates93] Bates, S. & Horwitz, S. (1993), 'Incremental Program Testing Using Program Dependence Graphs', in 'Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages'

[Bible01] Bible, J.; Rothermel, G. & Rosenblum, D. (2001), 'A Comparative Study of Coarse- and Fine-Grained Safe Regression Test Selection Techniques', *ACM Transactions on Software Engineering and Methodology*.

[Binkley95] Binkley, D. (1995), Reducing the cost of regression testing by semantics guided test case selection, *in* 'Proc. International Conference on Software Maintenance', pp. 251--260.

[Binkley99] Binkley, D. (1999), 'The Application of Program Slicing to Regression Testing', Master's thesis, Loyola College in Maryland.

[Bourque07] Bourque Fortin, M. (2007), 'Génération et exécution de séquences de tests unitaires pour les programmes orienté aspect', Master's thesis, Universite du Quebec a Trois-Rivieres.

[Briand02] Briand, L. C.; Labiche, Y. & Soccar, G. (2002), Automating impact analysis and regression test selection based on UML designs, *in* 'Proc. International Conference on Software Maintenance', pp. 252--261.

[Chen94] Chen, Y.-F.; Rosenblum, D. S. & Vo, K.-P. (1994), TESTTUBE: a system for selective regression testing, *in* 'Proc. ICSE-16. th International Conference on Software Engineering', pp. 211--220.

[CMSC04] 'CMSC 631 — Program Analysis and Understanding Fall 2004 (Course material)', http://www.cs.umd.edu/class/fall2002/cmsc631/

[Erlikh00] Erlikh, L. (2000), 'Leveraging legacy system dollars for e-business', *IT Professional* **2**(3), 17--23.

[Forgacs] Forgacs, I.; Hajnal, A. & Takacs, E. (1998), Regression slicing and its use in regression testing, *in* 'Proc. Twenty-Second Annual International Computer Software and Applications Conference COMPSAC '98', pp. 464--469.

[Galland02] Galland, S. (2002), 'Introduction aux diagrammes de collaboration', 'Matériel de cours Analyse Conception Objets', 'École Nationale Supérieure des Mines de Saint-Étienne', http://www.emse.fr/~boissier/enseignement/aco/.

[Graves98] Graves, T. L.; Harrold, M. J.; Kim, J.; Porters, A. & Rothermel, G. (1998), An empirical study of regression test selection techniques, *in* 'Proc. (20th) International Conference on Software Engineering', pp. 188--197.

[Gupta92] Gupta, R.; Harrold, M. J. & Soffa, M. L. (1992), An approach to regression testing using slicing, *in* 'Proc. Conference on Software Maintenance Proceerdings', pp. 299--308.

[Harrold01] Harrold, M.-J.; Jones, J.; Li, T.; Liang, D.; Orso, A.; Pennings, M.; Sinha, S. & Spoon, A. (2001), Regression Test Selection for Java Software, *in* 'OOPSLA'01'.

[JavaCC01] 'JavaCC et JJTree', https://javacc.dev.java.net/

[JUnit01] 'JUnit', http://www.junit.org

[Kim00] Kim, J.-M.; Porter, A. & Rothermel, G. (2000), An empirical study of regression test application frequency, *in* 'Proc. International Conference on Software Engineering', pp. 126--135.

[Korel98] Korel, K. & Al-Yami, A. (1998), 'Automated Regression Test Generation', in 'Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis'.

[Korel02] Korel, B.; Tahat, L. H. & Vaysburg, B. (2002), Model based regression test reduction using dependence analysis, *in* 'Proc. International Conference on Software Maintenance', pp. 214--223.

[Kung93] Kung, D.; Gao, J. & Hsia, P. (1995), 'Class firewall, Test Order and Regression Testing of Object Oriented Programs', in 'Journal of Object Oriented Programming', pp. 51 -- 65.

[Larman04] Larman, C.2eme, ed. (2004), *UML et les Design Patterns*, CampusPress.

[Lehman80] Lehman, M. M. (1980), 'Programs, life cycles, and laws of software evolution', *#IEEE_J_PROC#* **68**(9), 1060--1076.

[Li99] Li, Y. & Wahl, N. (1999), 'An Overview of Regression Testing', *Software Engineering Notes* **24**, 69-73.

[Massicotte06] Massicotte, P. (2006), 'Test orienté aspect: Une approche formelle basée sur les diagrammes de collaboration', Master's thesis, Universite du Quebec a Trois-Rivieres.

[Pilskalns06] Pilskalns, O.; Uyan, G. & Andrews, A. (2006), Regression Testing UML Designs, *in* 'Proc. 22nd IEEE International Conference on Software Maintenance ICSM '06', pp. 254--264.

[Ren05] Ren, X.; Ryder, B. G.; Stoerzer, M. & Tip, F. (2005), Chianti: a change impact analysis tool for Java programs, *in* 'Proc. 27th International Conference on Software Engineering ICSE 2005', pp. 664--665.

[Rothermel94] Rothermel, G. & Harrold, M. J. (1994), A framework for evaluating regression test selection techniques, *in* 'Proc. ICSE-16. th International Conference on Software Engineering', pp. 201--210.

[Rothermel96] Rothermel, G. & Harrold, M. J. (1996), 'Analyzing regression test selection techniques', *#IEEE_J_SE#* **22**(8), 529--551.

[Rothermel97-1]Rothermel, G. & Harrold, M. J. (1997), 'A Safe, Efficient Regression Test Selection Technique', *ACM Transaction on Software engineering and Methodology* **6**, 173-210.

[Rothermel97-2]Rothermel, G. & Harrold, M. J. (1997), 'Experience With Regression Test Selection', *Empirical Software Engineering Journal* **2**, 178-187.

[Rothermel00] Rothermel, G.; Harrold, M. J. & Dedhia, J. (2000), 'Regression Test Selection for C++ Software', *Journal of Software Testing, Verification and Reliability* **10**.

[Rothermel02] Rothermel, G.; Elbaum, S.; Malishevsky, A.; Kallakuri, P. & Davia, B. (2002), The impact of test suite granularity on the cost-effectiveness of regression testing, *in* 'Proc. 24rd International Conference on Software Engineering ICSE 2002', pp. 130--140.

[Skoglund05] Skoglund, M. & Runeson, P. (2005), A case study of the class firewall regression test selection technique on a large scale distributed software system, *in* 'Proc. International Symposium on Empirical Software Engineering', pp. 10pp..

[White05] White, L.; Jaber, K. & Robinson, B. (2005), Utilization of extended firewall for object-oriented regression testing, *in* 'Proc. 21st IEEE International Conference on ICSM'05 Software Maintenance', pp. 695--698.

[Wikipedia01] 'Analyse statique de code', http://en.wikipedia.org/wiki/Static_code_analysis.

[Wikipedia02] 'Analyseur syntaxique', http://fr.wikipedia.org/wiki/Parseur

[Wikipedia03] 'Arbre syntaxique abstrait', http://en.wikipedia.org/wiki/Abstract_syntax_tree

[Wu99] Wu, Y.; Chen, M.-H. & Kao, H. M. (1999), Regression testing on object-oriented programs, *in* 'Proc. 10th International Symposium on Software Reliability Engineering', pp. 270--279.

[Xu03] Xu, L.; Xu, B.; Chen, Z.; Jiang, J. & Chen, H. (2003), Regression testing for Web applications based on slicing, *in* 'Proc. 27th Annual International Computer Software and Applications Conference COMPSAC 2003', pp. 652--656.