Assemblage testable et validation de composants

Benoit Baudry

▶ **To cite this version:**

Benoit Baudry. Assemblage testable et validation de composants. Génie logiciel [cs.SE]. Université Rennes 1, 2003. Français. NNT : . tel-03341051

HAL Id: tel-03341051
https://theses.hal.science/tel-03341051

Submitted on 10 Sep 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre 2846

THÈSE

présentée

DEVANT L'UNIVERSITÉ DE RENNES 1

pour obtenir

le grade de : *DOCTEUR DE L'UNIVERSITE DE RENNES 1*

Mention : Informatique

par

Benoit Baudry

Équipe d'accueil : IRISA/TRISKELL
École doctorale : Matisse
Composante universitaire : IFSIC

Titre de la thèse :

Assemblage testable et validation de composants

Soutenue le 24 juin 2003 devant la commission d'Examen

COMPOSITION DU JURY :

M. :	Daniel	HERMAN	Président
MM. :	Stéphane	DUCASSE	Rapporteurs
	Bruno	LEGEARD	
	Bertrand	MEYER	
MM. :	Jean-Marc	JEZEQUEL	Examinateurs
	Yves	LE TRAON	

Remerciements

Je tiens tout d'abord à remercier Yves Le Traon d'avoir encadré cette thèse, et de m'avoir permis d'appréhender l'aspect captivant du travail de recherche au long de ces trois années. Je remercie Jean-Marc Jézéquel de m'avoir aidé à prendre conscience que ce travail s'inscrit dans un contexte global.

Je remercie Daniel Herman qui m'a fait l'honneur de présider le jury, ainsi que Bruno Legeard, Bertrand Meyer et Stéphane Ducasse qui ont accepté d'évaluer ce travail, et dont les remarques furent grandement appréciées.

Je remercie également Lionel Briand, Yvan Labiche, et Gerson Sunyé pour les échanges enrichissants que nous avons eus au cours de ces travaux.

Je souhaiterais aussi remercier tous ceux qui font que l'IRISA est un endroit où j'ai particulièrement apprécié de travailler. Franck pour le débat, la dynamique de win, la relecture et surtout pour toutes les choses extrêmement utiles qu'il a réalisées. Clémentine parce qu'il est bene de travailler avec quelqu'un qui parle italien, Tewfik qui voit ce que je veux dire, Loïc pour avoir supporté les crises de guerre des étoiles et tout le monde dans l'équipe Triskell. Merci aussi à Christelle, Marie-Noëlle, Nadège, Angélique, et aux filles de la cafet.

Je remercie Yannic Le Flem d'avoir dessiné l'affiche de thèse sans oublier le petit booléen vert pour Sandrine et Isabelle pour avoir su synthétiser cette thèse de façon remarquable.

Enfin, je remercie tous ceux qui m'ont permis de ne pas basculer du côté obscur de la thèse en faisant de la vie ce qu'elle ne doit jamais cesser d'être : passionnée. Je remercie tout d'abord Sandrine, pour la correspondance mythique qui a accompagné la rédaction de ce mémoire. Je remercie Céline, Corentin, Jérôme, Antoine, Anne-Gaëlle, Nolwenn, Johanne, François et Cécile d'avoir partagé ma vie au cours de ces trois années, parce que je crois à la cohérence de tout ça, et ce travail existe aussi grâce à eux. Je remercie Jean-Louis Sechet, Mathieu Saglio, Kerwin Roland, Nusch Werchowska, John Cage, John Coltrane, Cecil Taylor, Olivier Messiaen et Gÿorgi Ligeti dont la musique m'a toujours accompagné et aidé à avancer.

Je remercie mon grand-père Lou et mes parents, pour m'avoir soutenu tout au long de cette thèse, et pour avoir insufflé la vie aux cas de test.

ff

Extrait du **Regard de l'Église d'amour**, n°20 des **Vingt Regards sur l'Enfant Jésus**. Olivier Messiaen, 1947.

1 Introduction *1*

2 Etat de l'art *5*

2.1 Le test de logiciel **6**
2.1.1 Le test classique 6
2.1.2 le test de logiciels orientés objet 8
2.1.3 L'analyse de mutation 10
2.1.4 Conclusion 12

2.2 Éléments de la notation UML pour le test de logiciels OO **13**
2.2.1 Diagramme de classes 13
2.2.2 Vues dynamiques : diagrammes de séquence et d'états 15

2.3 Méthodologie pour une conception testable **16**
2.3.1 La conception par contrat 17
2.3.2 Les contrats et UML : le langage OCL 18
2.3.3 Les composants autotestables 19
2.3.4 Conclusion 21

2.4 La génération automatique de cas de test **22**
2.4.1 Techniques fonctionnelles 22
2.4.2 Techniques structurelles 23

2.5 Conception par contrat et test **25**

2.6 Testabilité et mesures pour les logiciels OO **27**

2.7 Conclusion **29**

3 Génération automatique de cas de test pour un composant *31*

3.1 Adaptation de l'analyse de mutation pour la qualification de composants **32**
3.1.1 Le processus global 33
3.1.2 L'oracle pour l'analyse de mutation 34
3.1.3 Mutants équivalents 36
3.1.4 Opérateurs de mutation utilisés pour la qualification des composants 37
3.1.5 Outils 38
3.1.6 Analyse de mutation pour une classe ou un système 39
3.1.7 Optimisation automatique et analyse de mutation 41

3.2 Algorithme génétique pour l'optimisation de cas de test **42**
3.2.1 Les algorithmes génétiques 42
3.2.2 Texte de l'algorithme 44
3.2.3 Le problème de l'optimisation de cas de test 46

3.3 Etudes de cas pour un algorithme génétique **51**

3.3.1 Optimisation de cas de test unitaires : un exemple en Eiffel 51
3.3.2 Optimisation de cas de test système : l'exemple d'un composant .NET 53
3.3.3 Résultats et remise en cause du modèle génétique 54

3.4 Une approche adaptative : un « algorithme bactériologique » 57
3.4.1 Le modèle bactériologique 57
3.4.2 De nouveaux résultats 61
3.4.3 Discussion et validation du modèle bactériologique 62

3.5 Paramétrage des modèles 64
3.5.1 Paramétrage du modèle bactériologique 64
3.5.2 Recherche d'un algorithme intermédiaire 66

3.6 Conclusion 68

4 Robustesse et diagnosabilité : impact de la conception par contrat sur un assemblage de composants 69

4.1 Conception par contrat et élaboration d'une mesure 70
4.1.1 Les contrats pour la robustesse et la diagnosabilité 70
4.1.2 Elaboration d'une mesure 71

4.2 Mesure de la robustesse 72
4.2.1 Définitions 72
4.2.2 Axiomatisation 74
4.2.3 Hypothèses et modèle mathématique 76
4.2.4 Démonstration des axiomes 77
4.2.5 Expériences pour paramétrer le modèle 82
4.2.6 Application de la mesure sur trois systèmes réels et résultats 87
4.2.7 Critique du modèle de mesure 89

4.3 Une mesure de la diagnosabilité 91
4.3.1 Analyse du problème 91
4.3.2 Définitions 93
4.3.3 La mesure de diagnosabilité 94
4.3.4 Résultats et conclusions 99

4.4 Conclusion 99

5 Anti-patterns de testabilité dans un assemblage de composants 101

5.1 Présentation de la problématique 101

5.2 Testabilité d'une architecture OO: définitions et méthodologie 104
5.2.1 Testabilité 104
5.2.2 Une méthodologie pour la conception de systèmes testables 105

5.2.3 Exemple 106

5.3 Critère de test et anti-patterns pour les architectures OO 107

5.3.1 Analyse informelle des anti-patterns de testabilité 107

5.3.2 Complexité de l'héritage 109

5.3.3 Critère de test pour des systèmes OO 110

5.3.4 Exemple pour la génération de test 113

5.4 Modélisation des anti-patterns 116

5.4.1 Construction d'un graphe à partir d'un diagramme de classes d'UML 116

5.4.2 Détecter des anti-patterns à partir du GDC 118

5.4.3 Complexité des anti-patterns 119

5.4.4 Mesure de la complexité des anti-patterns : système de gestion de livres 121

5.5 Amélioration de la testabilité du modèle 122

5.6 Exemples d'application 124

5.6.1 Le gestionnaire de livres 124

5.6.2 Serveur De Réunion Virtuelle 125

5.6.3 Une architecture de compilateur 127

5.7 Design Patterns pour la testabilité du modèle 129

5.7.1 Concevoir par cristallisation de patterns 129

5.7.2 Application du design pattern State pour améliorer la testabilité 131

5.8 Analyse de testabilité des design patterns 134

5.8.1 Analyse de testabilité de State 134

5.8.2 Analyse de testabilité d'Abstract Factory 135

5.8.3 Définir des contraintes de testabilité au niveau méta 136

5.8.4 La grille de testabilité des design patterns 138

5.9 Conclusion 140

6 Conclusions et perspectives 141

6.1 Contributions majeures 141

6.1.1 Validation de composants 141

6.1.2 Testabilité d'un assemblage de composants 143

6.2 Perspectives 143

6.2.1 La génération de test pour le diagnostic 143

6.2.2 Les contrats comme oracle de test 144

6.2.3 Transformation de modèles pour la testabilité 144

6.2.4 Design patterns et test 144

Annexes ____ *145*

Annexe A *Répartition des contrats dans un système OO* ____ *147*

Annexe B *Interactions d'objets pour le gestionnaire de livres* ____ *151*

AU1 ____ **151**

AU2 ____ **152**

IC(BOOKEVENT, BOOK) ____ **153**

Glossaire ____ *155*

Bibliographie ____ *159*

1
Introduction

Le *test de logiciel* apparaît aujourd'hui comme le moyen principal pour la validation du fonctionnement d'un programme [Beizer"90; Binder"99]. Il a pour objectif d'examiner ou d'exécuter un programme dans le but d'y révéler des erreurs. Il est souvent défini comme le moyen par lequel on s'assure qu'une implantation est conforme à ce qui a été spécifié. L'activité de test est omniprésente tout au long du cycle de vie du logiciel, et différentes techniques permettent de valider les différentes étapes du développement.

Par ailleurs, l'industrie du logiciel produit aujourd'hui des systèmes de plus en plus complexes ; il devient donc crucial de factoriser le savoir-faire et les produits. Le savoir-faire est réutilisé sous la forme de processus et de méthodologies pour la gestion de projets logiciels ou par la mutualisation de solutions éprouvées au niveau de la conception grâce, en particulier, à la notion de *design patterns* [Gamma"95]. Le développement de *canevas d'application* (« frameworks ») et de *composants logiciels* constitue une des réponses à la réutilisation de certaines parties de l'implantation.

La programmation orientée objets (OO) offre une solution élégante au développement de composants réutilisables comme unité de déploiement [Szyperski"98]. Elle offre en particulier des possibilités d'encapsulation et de masquage d'information qui permettent d'établir une analogie avec les composants matériels, et la notion d'adaptabilité qui permet de les adapter souplement. Ce type de programmation est maintenant largement utilisé pour l'analyse, la conception et la réalisation de grands systèmes d'information. L'adoption des langages orientés objets s'est accompagnée de l'émergence d'UML (Unified Modeling Language) comme langage standard de modélisation. Ce langage permet la description des différents aspects du logiciel, de la description structurelle et du comportement dynamique du programme jusqu'à la définition des cas de test ou de l'environnement de déploiement.

L'adoption généralisée du paradigme objet, ainsi que le changement d'échelle pour le développement de logiciel fait émerger la nécessité de méthodes adaptées pour le test

[Binder"99]. En effet, le besoin de fiabilité et de robustesse pour les composants est d'autant plus grand que ceux-ci vont être réutilisés dans de nombreux contextes différents. L'encapsulation des données dans des classes, la répartition du contrôle à travers le système, ou la liaison dynamique sont autant de spécificités qui doivent être prises en compte par les techniques et méthodes de test proposées. Par ailleurs, il semble intéressant d'appuyer ces méthodes sur UML comme langage de modélisation des logiciels.

Comme nous le verrons par la suite, il existe de très nombreux travaux qui se sont intéressés au test de logiciels orientés objet, allant du test d'unités indépendantes jusqu'aux processus génériques pour la validation de systèmes complexes. Ceci a donné lieu, par exemple, à une standardisation du test de classe, à travers la popularisation d'une famille de frameworks pour divers langages orientés objet appelée Xunit [Beck"01; Craig"02]. Ces frameworks prennent en compte la structure particulière des programmes OO pour l'écriture, l'exécution et la collecte des résultats du test pour une classe. Par ailleurs, de la même manière que les design patterns et les composants permettent la réutilisation d'architectures et de code, il existe des patterns de test qui correspondent à des définitions abstraites de plans de test ou de cas de test qui peuvent être réutilisées pour différentes applications.

Les travaux présentés dans cette thèse s'articulent autour de trois contributions majeures qui s'inscrivent dans le cadre global du test de logiciels orienté objet pour le développement de composants logiciels fiables.

Considérant que le test constitue le moyen principal pour l'évaluation de la fiabilité d'un composant, la génération de cas de test efficaces est un problème important. Or, s'il est possible de générer rapidement des cas de test couvrant les utilisations normales d'un programme, l'amélioration de cet ensemble pour qu'il soit efficace dans tous les cas est très coûteuse à la fois en temps et en effort. **L'optimisation automatique d'un ensemble de cas de test est donc la première contribution de cette thèse**. Ce travail a consisté à étudier un *algorithme génétique*, et à définir une nouvelle catégorie d'algorithmes évolutionnistes, appelée *algorithme bactériologique*, pour améliorer automatiquement la qualité d'un ensemble de cas de test pour un composant. Cet aspect de nos travaux a été publié dans [Baudry"00a; Baudry"00b; Baudry"00c; Baudry"02b; Baudry"02c; Baudry"02d].

Dans la suite, nous étudions des problèmes de test qui peuvent apparaître lors de l'assemblage de composants. Cette phase particulière du test est appelée test d'intégration et consiste à tester les interactions entre les composants du système. A ce niveau, deux types d'erreur peuvent apparaître : des erreurs dans la définition des interfaces ou des incohérences dans le système dues à la décentralisation du contrôle.

Le premier type d'erreur peut entraîner une mauvaise utilisation d'un composant par un autre. Pour éviter ces erreurs, la spécification des interfaces doit être la plus complète

possible. Cette spécification pouvant être exprimée sous forme de contrats dans un cadre OO, **la seconde contribution majeure présentée dans cette thèse consiste à étudier l'impact des contrats sur deux facteurs de qualité d'un logiciel : la robustesse** (capacité d'un logiciel à détecter une erreur interne) **et la diagnosabilité** (facilité pour la localisation d'une erreur) [Baudry"01a; Le Traon"03].

Le second type d'erreur pouvant se produire lors de l'intégration est dû au contrôle largement réparti à travers le système dans les logiciels orientés objet. En effet, l'utilisation importante de certains mécanismes objets tels que le polymorphisme et la délégation, entraîne l'emploi d'un grand nombre d'objets pour exécuter une fonctionnalité. Dans ce cas, des erreurs peuvent se produire lors de l'utilisation de certains objets distants du système par d'autres objets. **La troisième point abordé au cours de cette thèse est donc l'étude d'un critère de test pour assurer l'intégration d'un assemblage de composants, ainsi qu'une mesure de complexité associée à ce critère permettant de prévoir la testabilité d'un assemblage**. Les études de la testabilité de systèmes orientés objet ont été publiées dans [Baudry"01b; Baudry"02a; Baudry"03].

Ces trois contributions peuvent être classées suivant deux dimensions indépendantes et complémentaires : soit dans une perspective conception/validation, soit par un découpage composant/assemblage de composants.

Le premier aspect (conception/validation) consiste à répartir les travaux entre ceux qui traitent de la validation (côté « curatif ») et ceux qui traitent de la conception (côté « préventif »). Dans une répartition suivant cet aspect, les deux premières contributions peuvent être regroupées dans la catégorie validation, puisqu'elles concernent l'étude des *composants autotestables* (approche pragmatique préconisée dans l'équipe Triskell). La troisième contribution, concerne l'étude de propriétés d'un système qui peuvent être observés dés les premières phases de la conception (côté « préventif »).

Le second aspect (composant/assemblage) réparti les travaux entre ceux qui se concentrent sur un composant unitaire et ceux qui étudient le comportement d'un assemblage de composants. Avec cette seconde répartition, la première contribution apparaît seule, puisqu'elle concerne la génération et l'optimisation automatique de cas de test pour un composant isolé. Par ailleurs, les deux autres contributions concernant l'étude du comportement d'un assemblage de composants vis-à-vis du test, sont alors regroupées suivant cet aspect.

Ainsi, il apparaît que, même si ces contributions sont clairement trois facettes d'un même problème, elles peuvent être présentées suivant différents points de vue, rendant l'articulation de ce document parfois difficile. Dans la suite, nous essayons d'être le plus précis possible, et de resituer, si nécessaire, les différents points en fonction de ces deux aspects.

La suite de ce document est organisée de la manière suivante :

Le chapitre 2 rappelle les principes généraux pour le test de logiciel, puis présente l'état de l'art du test pour les programmes orientés objet. Au cours de ce chapitre, nous introduisons aussi une méthode pour la conception de composants logiciels fiables : les composants autotestables. Enfin, nous détaillons les travaux existant sur les points précis abordés dans cette thèse : la génération automatique de test, les assertions pour le test de logiciel et la testabilité.

Le chapitre 3 se concentre sur l'étude d'un algorithme génétique pour résoudre le problème de l'optimisation et la génération automatique de cas de test. Les résultats expérimentaux montrant une convergence trop lente, et discontinue, nous proposons un algorithme original, mieux adapté à la génération d'un ensemble de cas de test. Nous appelons cet algorithme un algorithme bactériologique, et validons son efficacité expérimentalement.

Le chapitre 4 concerne l'étude d'un autre aspect de la méthode : la conception par contrat (*design by contract*). Nous étudions ici l'impact de cette technique pour la détection et la localisation d'erreur. Les mesures de robustesse et de diagnosabilité sont définies pour évaluer cet impact, et les facteurs influençant ces mesures sont calibrés par plusieurs expériences. Les résultats valident à la fois l'approche par contrat et l'intérêt des composants autotestables pour augmenter la confiance dans le composant.

Le chapitre 5 présente le dernier point étudié dans cette thèse : la testabilité d'un assemblage de composants. Nous proposons un critère de test pour la couverture d'interactions entre objets, ainsi qu'une mesure de complexité associée. Ce critère peut être évalué sur un diagramme de classes, et permet ainsi d'avoir une mesure prédictive sur la qualité de l'implantation. Nous présentons aussi une méthode pour l'amélioration de la testabilité d'un programme à partir du diagramme de classes. Enfin, au cours de cette étude nous nous concentrons sur les design patterns comme un sous-ensemble cohérent dans un diagramme de classes pour l'analyse de testabilité à un niveau local. Nous proposons une analyse détaillée de la testabilité de l'application de designs patterns ; quels problèmes classiques de testabilité peuvent être résolus, et quels sont les nouveaux points complexes pour le test.

Enfin, le chapitre 6 conclut cette thèse et trace un panorama des principales pistes de recherche et perspectives qui se dégagent à partir de ces travaux.

2
Etat de l'art

La programmation orientée objets a introduit de nouvelles structures qui doivent être prises en compte pour le test de logiciels. L'encapsulation des données dans des classes, l'héritage, le polymorphisme, font l'essentiel des constructions qui obligent à repenser le test de logiciels orientés objet(OO). De plus la construction des logiciels OO rend le découpage méthodologique des phases de test en unité-intégration-système inadapté dans la majeure partie des cas. Par exemple, certaines unités dans un système sont trop liées les une aux autres pour pouvoir être testées séparément, et on peut considérer la construction d'un système à objets comme une succession d'étapes d'intégration.

Depuis dix ans, de nombreux travaux se sont intéressés au test de logiciels OO, et récemment plusieurs ouvrages ont abordé ce sujet dans sa globalité [Siegel"96; Binder"99; McGregor"01]. Les premiers travaux ont consisté à appliquer les techniques du test de logiciels procéduraux au test de logiciels OO en n'apportant que quelques légères modifications (par exemple en prenant la classe comme unité de test et non plus la procédure/méthode). Avec l'apparition de standards méthodologiques (design by contract, rational process, catalysis…) ou de notation (UML) pour la construction de logiciels OO, la recherche sur le test de logiciels OO a pris en compte certaines spécificités de la programmation OO et des techniques originales sont apparues. Si ces travaux ont tenu compte très tôt du polymorphisme et de l'utilisation d'assertions/contrats pour le test, le choix de la notation UML (Unified Modeling Language) comme support de définition de test est relativement récent et l'analyse du processus même de conception OO en vue du test (patrons de conception, contrôle distribué) est quasiment inexistante.

Le but de ce chapitre est d'introduire le domaine du test, d'abord d'un point de vue général, puis de détailler les points particuliers correspondant aux trois contributions de cette thèse. Nous commençons par de brefs rappels sur l'activité du test de logiciel et le vocabulaire associé, puis nous présentons les travaux spécifiques au test de logiciels orientés objet, enfin nous terminons la première section avec la présentation d'une technique de validation de cas de test très largement utilisée au cours des travaux décrits dans cette thèse : l'analyse de mutation. Nous présentons ensuite quelques éléments de la notation UML utilisés pour le test, et continuons avec l'introduction d'une méthode pour la conception de composants logiciels fiables. Enfin, les trois dernières sections présentent des états de l'art détaillés relatifs aux trois points abordés dans la suite de cette thèse : la génération automatique de cas de test, l'impact de la conception par contrat pour le test et la testabilité des logiciels.

2.1 Le test de logiciel

Dans cette première section, nous présentons l'approche générale pour le test de logiciels, puis nous détaillons les travaux particuliers au test de logiciels orientés objet. Enfin, nous introduisons l'analyse de mutation, une technique particulière pour la validation et la génération de cas de test qui est utilisée dans les chapitre 3 et 4.

2.1.1 Le test classique

Le test de logiciel a pour but de détecter des erreurs dans un programme. Les termes de faute, erreur et défaillance ont été définis précisément dans [Laprie"95]. Une *faute* désigne la cause d'une erreur, une *erreur* est la partie du système qui est susceptible de déclencher une défaillance, et une *défaillance* correspond à un événement survenant lorsque le service délivré dévie de l'accomplissement de la fonction du système. Dans la suite de ce chapitre nous n'utiliserons que le terme erreur pour désigner le but du test.

Très schématiquement, le test dynamique de logiciel consiste à exécuter un programme avec un ensemble de données en entrée, et à comparer les résultats obtenus avec les résultats attendus. Si les résultats diffèrent, une erreur a été détectée, auquel cas, il faut localiser cette erreur et la corriger. Quand l'erreur est corrigée, il convient de retester le programme pour s'assurer de la correction et la non régression du programme. Enfin, pour être complète, l'activité de test implique de déterminer un « critère d'arrêt », qui spécifie quand le programme a été suffisamment testé.

A chacune de ces étapes pour le test correspond un terme particulier. Pour fixer la terminologie, et en s'inspirant de [Xanthakis"92], on considère que le programme testé est appelé *programme sous test* que les *données de test* désignent les données en entrée du programme sous test, et qu'un *cas de test* désigne le programme chargé d'exécuter le programme sous test avec une donnée de test particulière. Le cas de test se charge aussi de mettre le programme sous test dans un état approprié à l'exécution avec les données de test en entrée. Par exemple, pour tester le retour d'un livre dans une bibliothèque, il

faut d'abord emprunter le livre. Au cours du chapitre 3, nous nous intéressons au problème de la génération automatique de données de test efficaces. L'efficacité de ces données est évaluée grâce à l'*analyse de mutation* qui est présentée dans la suite de ce chapitre.

Après l'exécution de chaque cas de test, il faut comparer le résultat obtenu avec le résultat attendu. Ce prédicat qui permet de déterminer si le résultat est incorrect est appelé un *oracle* ou une *fonction d'oracle*. Ce prédicat peut être explicite dans les cas de test, être obtenu indirectement par des assertions ou d'autres moyens (par exemple, le model-checking) ou, dans le cas le pire, être implicite et dépendre d'un verdict humain. Il est clair que l'oracle peut être plus ou moins spécifique à la donnée de test, et pourra ne pas détecter une erreur. Par contre, on attend de l'oracle d'être correct, c'est-à-dire de ne pas marquer comme erroné un comportement correct. L'oracle est donc le reflet, plus ou moins complet, et exécutable, de la spécification.

Lorsqu'un cas de test échoue, il faut localiser la source de la défaillance dans le programme pour pouvoir corriger la faute. Cette étape de localisation est appelée la *phase de diagnostic*. Lorsque le résultat obtenu est conforme à l'oracle, la dernière étape du test consiste à déterminer si les cas de test sont suffisant pour garantir la qualité du logiciel. Pour cela, il faut définir un *critère de test* ou *critère d'arrêt* et vérifier si les cas de test générés vérifient ou non ce critère. Les techniques de génération visent donc souvent à vérifier un critère d'arrêt particulier (plutôt qu'à chercher à détecter des erreurs). Pour générer des cas de test en fonction d'un critère de test, il est possible de définir des *objectifs de test*. Par exemple, si le critère exige l'exécution de toutes les instructions du programme au cours du test, « couvrir l'instruction 11 » est un objectif de test possible. Il faut ensuite écrire un cas de test qui vérifie cet objectif. Cette notion d'objectif de test s'applique non seulement aux aspects structurels (couverture de code), mais aussi aux aspects comportementaux (observation d'un certain échange de messages [Pickin"02]). Le chapitre 5 définit un critère de test pour les logiciels OO, et s'en sert comme support pour une analyse prédictive de la testabilité visant à estimer l'effort de test.

Les techniques pour la génération de cas de test sont de deux types : le *test fonctionnel* et le *test structurel* [Beizer"90]. Si le programme sous test est considéré comme une boîte noire (on ne tient pas compte de la structure interne du programme), on parle de test fonctionnel. Dans ce cas, la génération de cas de test se fait à partir d'une spécification la plus formelle possible du comportement du programme. Cette technique a pour avantage de générer des cas de test qui seront réutilisables même si l'implantation change ; elle ne garantit cependant pas que tout le programme ait été couvert par les cas de test.

Le test structurel s'appuie sur la structure interne du programme pour générer des cas de test. Les techniques de test structurel se basent généralement sur une représentation

abstraite de cette structure interne, un *modèle* du programme, très souvent un graphe (graphe flot de contrôle [Beizer"90], graphe flot de données [Rapps"85]). Cette représentation permet de définir des critères de couverture indépendants d'un programme particulier : couverture de tous les arcs du graphe, tous les nœuds, tout ou partie des chemins… Le test structurel a pour avantage de permettre de valider les cas de test générés, en fonction de leur taux de couverture du critère.

Le test a lieu à différents moments dans le cycle de développement du logiciel, les différentes étapes sont traditionnellement les suivantes :

- le test unitaire : une unité est la plus petite partie testable d'un programme, c'est souvent une procédure ou une classe dans les programmes à objet.
- le test d'intégration : consiste à assembler plusieurs unités et à tester les erreurs liées aux interactions entre ces unités (et éventuellement à détecter les erreurs rémanentes non détectées au niveau unitaire).
- le test système : teste le système dans sa totalité en intégrant tous les sous-groupes d'unités testés lors du test d'intégration. Il s'étend souvent à d'autres aspects tels que les performances, le test en charge…

2.1.2 le test de logiciels orientés objet

Les mécanismes propres aux langages orientés objet et les méthodes d'analyse de conception et de développement associées, ont entraîné la nécessité de nouvelles techniques de test pour les logiciels fondés sur ces langages et méthodes. Une première modification a été le changement d'échelle pour le test unitaire. En effet, cette partie du test se concentrait sur les procédures dans le cadre des langages procéduraux, alors qu'elle s'intéresse à une classe dans un cadre orienté objet.

Il existe de nombreux travaux sur le *test de classe*, concernant les différents problèmes que sont la génération de données de test, les critères de couverture, la production d'un oracle, et l'écriture des drivers de test. En ce qui concerne les critères de couverture, [Harrold"94; Buy"00] étudient les critères flot de données et flot de contrôle pour le test d'une classe. Dans [Harrold"94], Harrold et al. proposent trois niveaux de granularité (intra-méthode, inter-méthode et intra-classe) pour l'analyse des flots de données dans une classe. Ils donnent l'algorithme pour construire le graphe et l'illustrent sur une classe. Buy et al. [Buy"00] repartent des travaux de Harrold et étendent l'approche à des techniques d'exécution symbolique et de déduction automatique, dans le but de générer automatiquement des séquences d'appels de méthodes pour tester une classe. La combinaison de ces trois techniques pour la génération de test est illustrée en détail sur un exemple. Les auteurs ont étudié, à la main, la faisabilité de l'approche sur plusieurs études de cas, et veulent maintenant développer les outils pour l'automatiser. Quant à McGregor [McGregor"01], il propose trois critères fondés sur la couverture de la machine

à états associée à la classe, sur la résolution des paires de contraintes pré/post conditions, et la couverture de code.

Pour la génération des données de test plusieurs solutions ont été proposées. Dans [Buy"00], les auteurs s'intéressent à l'exécution symbolique, McGregor [McGregor"01], propose d'utiliser les contraintes les pré et post conditions exprimées en OCL pour déduire les données de test pertinents automatiquement. Dans la suite de cette thèse nous étudions deux algorithmes évolutionnistes pour la génération automatique de données de test pour un composant orienté objet ces travaux ont été publiés dans [Baudry"00b; Baudry"02d]). Les travaux qui se sont intéressés à la production de l'oracle pour le test unitaire de classes, sont tous fondés sur la spécification de post-conditions pour les méthodes de la classe [Edwards"01; Cheon"02; Fenkam"02]. Ces travaux sont détaillés dans la section 2.5 sur l'utilisation des contrats pour le test.

Enfin, pour l'organisation des cas de tests, et la production de drivers de test, [Jézéquel"01] propose d'embarquer les cas de test dans la classe sous test. C'est l'idée de *classe auto-testable* sur laquelle s'appuie en partie cette thèse, et qui sera détaillée dans la section suivante. D'autre part, la méthode dite d'eXtremeProgramming proposée par Kent Beck [Beck"99] préconise une utilisation importante du test unitaire. Dans ce cadre, une famille de frameworks a été développée pour le test unitaire de classes dans différents langages orientés objet : Junit pour Java, EiffelUnit pour Eiffel, Nunit pour la plate-forme .NET [Beck"01; Craig"02] … Une bonne structuration des cas de test unitaires offre aussi l'avantage d'une réutilisation facile au moment du test de non régression.

Un autre point particulier pour le test de logiciels orientés objet, est le *test d'intégration*. En effet, les systèmes OO peuvent être vus comme l'assemblage de plusieurs unités offrant chacune une interface aux autres unités dans le système. Si une unité utilise les services offerts par une autre unité, la première est appelée *unité cliente* et la seconde *unité serveur*. Le test d'intégration consiste alors à vérifier que les unités clientes utilisent les unités serveur en conformité avec l'interface offerte par l'unité serveur. Des critères de couverture pour le test des interactions entre classes, ou entre composants dans un système ont été proposés dans [Chen"99; Alexander"00; Alexander"02a; Baudry"02a; Martena"02; Wu"03]. Tous ces critères visent à couvrir certains types de relations entre des classes, dans le but de découvrir des erreurs dues à une mauvaise utilisation d'une unité serveur par une unité cliente. De plus, dans [Wu"03] les auteurs décrivent les éléments de spécification UML nécessaires au test d'intégration de composants. Wu et al. décrivent les descriptions comportementales minimum pour pouvoir tester les interactions avec un composant dont le code source n'est pas disponible. Les auteurs de [Hoijin"98; Ghosh"00] proposent d'utiliser des techniques de mutation sur les interfaces des composants à intégrer pour valider le test d'intégration.

Un problème complémentaire au test d'intégration, est d'obtenir un *ordre d'intégration* des composants d'un système qui soit efficace. En effet, les classes dans un

système orienté objet sont interdépendantes, et il peut apparaître des cycles dans ces relations de dépendance. Dans ce cas, il faut simuler le comportement d'une classe pour pouvoir en tester une autre. Or, l'écriture d'un simulateur peut être très coûteuse, et plusieurs travaux ont étudié des techniques pour trouver un ordre d'intégration qui minimise la génération de simulateurs [Kung"96; Tai"99; Labiche"00; Briand"01; Hanh"02]. Soundarajan et al. présentent une étude en rapport avec l'ordre d'intégration dans [Soundarajan"01]. Cet article est fondé sur le fait que l'héritage permet de développer des systèmes de manière incrémental, et que les cas de test pour les sous-classes doivent être développés aussi de manière incrémental à partir des cas de test pour les super-classes.

Pour le *test système*, plusieurs méthodes pour la production de cas de test ont été proposées [Briand"02a; Nebut"02; Pickin"02]. Ces trois méthodes sont fondées sur la conjonction d'informations obtenues à partir de plusieurs vues UML du système. Les auteurs de [Briand"02a] proposent d'utiliser les cas d'utilisation et les diagrammes de collaboration associés pour ordonner les cas de test dans une suite de tests système, puis d'utiliser les diagrammes de séquence ainsi que les pré et post conditions sur les méthodes pour dériver des cas de test. Dans [Nebut"02], l'idée est de définir des patrons de test abstraits sous forme de diagrammes de séquence génériques, qui peuvent être instanciés vis-à-vis d'un diagramme de classe particulier. Enfin [Pickin"02] s'appuie sur la machine à états associée à chaque classe du système, et une description sous forme de diagramme de séquence d'un objectif de test, pour générer un cas de test correspondant à l'objectif.

Enfin, certains travaux se concentrent sur certaines vues d'UML pour le test de logiciels OO. De nombreux travaux traitent de la génération de test à partir des machines à état, avec des diagrammes d'états UML [Kim"99; Offutt"99; Bogdanov"01; Chevalley"01b; Chevalley"01a; Antoniol"02]. Certains de ces travaux proposent des critères de couverture des diagrammes d'états, comme la couverture de tous les états, couverture des transitions ou encore couverture des paires de transition. Ces critères permettent alors la génération automatique de cas de test à partir d'un tel diagramme. Les auteurs de [Abdurazik"00] utilisent le diagramme de collaborations pour tester certaines propriétés des routines décrites à l'aide de ces diagrammes.

Dans la section suivante nous nous concentrons sur une technique pour évaluer la qualité des cas de test pour un composant : l'analyse de mutation. Cette méthode qui fut d'abord proposée pour les programmes procéduraux, a ensuite été adaptée pour le test de logiciels orientés objet.

2.1.3 L'analyse de mutation

L'analyse de mutation est une technique pour la validation de données de test fondée sur l'injection de fautes. L'insertion systématique d'erreurs dans un logiciel est souvent utilisée pour la validation expérimentale de techniques de test. La connaissance du

nombre et de l'emplacement précis des erreurs dans le logiciel permet en effet de contrôler l'efficacité de techniques ou méthodes de test ou de diagnostic [Jones"02].

L'analyse de mutation est une technique proposée par DeMillo [DeMillo"78] et peut être utilisée de deux façons : évaluer la qualité d'un ensemble de cas de test, guider la génération de test. Dans les deux cas la technique consiste à créer un ensemble de versions erronées du programme sous test, appelées *mutants*, et à exécuter un ensemble de cas de test sur chacun de ces programmes erronés. Un mutant est une copie exacte du programme sous test dans lequel une seule erreur simple a été injectée. En pratique, l'ensemble des mutants est créé en soumettant le programme à des *opérateurs de mutation* qui correspondent à différents types d'erreur (changement de signe des constantes, changement de sens d'une inégalité…).

Le résultat de l'exécution des cas de test avec les mutants permet d'une part d'évaluer l'efficacité de l'ensemble de cas de test par la proportion de mutants détectée (appelée le *score de mutation*). D'autre part, l'analyse des erreurs qui n'ont pas été détectées permet de guider la génération de nouveaux cas de test. Ceci consiste à couvrir les zones où se situent ces erreurs, et à générer des données de test spécifiques pour couvrir les cas particuliers. Un vocabulaire particulier est associé à cette approche : si un cas de test peut détecter un mutant, on dit qu'il *tue le mutant*, sinon le *mutant est vivant*.

Chaque mutant ne comporte qu'une seule erreur simple. Cette limitation est fondée sur deux hypothèses : *l'hypothèse du programmeur compétent*, et *l'effet de couplage*. La première hypothèse met en avant le fait qu'un programmeur compétent écrit des programmes « presque » corrects. Autrement dit, un programme écrit par tel programmeur est sûrement incorrect, mais il est « proche » de la version correcte (il suffit de modifications mineures pour le corriger). L'hypothèse sur l'effet de couplage dit que si un ensemble de tests peut détecter les fautes simples dans un programme, alors il pourra détecter des fautes plus complexes. Même si les notions de faute simple et faute complexe ne sont pas formalisées, Offutt propose la définition suivante dans [Offutt"92]:

Faut simple, faute complexe. *Une* faute simple*, est une faute qui peut être corrigée en ne modifiant qu'une seule instruction dans le code source. Une* faute complexe *est une faute qui ne peut pas être corrigée en ne modifiant qu'une seule instruction.*

Dans ce même article, Offutt étudie l'injection de plusieurs erreurs. Il exécute ensuite les cas de test efficaces avec mutants ne comportant qu'une seule erreur sur les mutants comportant plusieurs erreurs. Ces expériences montrent que des cas de test détectant des erreurs simples détectent aussi des erreurs plus complexes issues de la combinaison de plusieurs erreurs simples, ce qui tend à valider l'hypothèse sur l'effet de couplage.

Offutt étudie aussi l'efficacité de l'analyse de mutation en la comparant à des critères flot de donnée dans [Offutt"96a]. Dans cet article, Offut compare les cas de test générés

avec les deux techniques de deux manières : il regarde si les cas de test générés pour un critère peuvent satisfaire l'autre critère, et il compare le pouvoir de détection d'erreur de chacun. Il conclut que l'effort nécessaire pour la génération est identique dans les deux cas, mais que les cas de test générés par mutation satisfont plus facilement les critères flot de donnée, et découvrent plus d'erreurs.

Les erreurs injectées pour l'analyse de mutation correspondent à différents types d'erreurs de programmation. Ces types ont été isolés en observant les pratiques des programmeurs et en en analysant les erreurs détectées lors du test. De nombreux opérateurs ont été proposés pour les langages procéduraux, et Offutt étudie le sous-ensemble suffisant dans [Offutt"96b].

La génération de cas de test fondée sur l'analyse de mutation est fastidieuse, puisque tant qu'une proportion importante de mutants n'est pas détectée par les cas de test, il faut regarder pourquoi ils ne sont pas détectés, puis produire un cas de test visant à détecter chacun d'entre eux. Dans [DeMillo"91], les auteurs donnent une technique pour la génération automatique de cas de test fondée sur l'analyse de mutation. L'idée qu'ils développent dans cet article, et qu'ils ont affinée par la suite, est de fixer comme objectif de test l'endroit du programme où se trouve l'erreur, et de remonter toutes les contraintes sur les données entre cet endroit et l'entrée du programme. Ils peuvent ainsi générer une donnée de test qui atteint l'erreur et permet de détecter le mutant. Un outil pour le langage Fortran a été développé [DeMillo"93].

L'analyse de mutation a d'abord été conçue pour le test de logiciels procéduraux, mais depuis plusieurs années, des travaux se sont intéressés à cette technique dans le cadre de logiciels OO [Hoijin"98; Ghosh"00; Chevalley"01c; Kim"01; Alexander"02b; Ma"02]. Certains proposent de nouveaux opérateurs de mutation fondés sur les mécanismes spécifiques aux langages orientés objet [Chevalley"01c; Kim"01; Alexander"02b; Ma"02], alors que d'autres utilisent cette technique pour valider des cas de test d'intégration [Hoijin"98; Ghosh"00] ou unitaires [Baudry"00b]. Enfin, dans [Chevalley"01c] les auteurs détaillent une technique de génération automatique pour des mutants de programmes JAVA fondée sur le mécanisme d'introspection du langage.

2.1.4 Conclusion

Le test est une technique importante pour la validation de logiciels, et il existe de nombreux travaux qui s'intéressent aux différents aspects que recouvre cette activité. Dans cette section, nous avons présenté les principes et techniques principaux pour le test et avons introduit les différentes activités existantes autour du test de logiciels orientés objet. Enfin, nous avons introduit l'analyse de mutation qui est largement utilisée dans la suite de cette thèse pour évaluer la qualité des cas de test pour un programme.

Les deux sections suivantes illustrent certaines spécificités de la conception de logiciels orienté objet. Nous résumons tout d'abord quelques éléments de la notation UML qui seront nécessaires dans la suite pour illustrer certaines spécificités des

programmes orientés objet. Ensuite, la section 2.5 détaille la méthode de conception par contrat, ainsi qu'une méthode plus générale pour la conception de composants logiciels fiables.

2.2 Éléments de la notation UML pour le test de logiciels OO

Nous définissons ici les principaux éléments de la notation UML [Fowler"97; Booch"99] qui sont utilisés pour le test de logiciels OO. UML propose neuf diagrammes différents, aussi appelés *vues*. Nous présentons les diagrammes de classes, de séquence, d'activité et d'état. Il existe plusieurs travaux sur le test OO fondés sur ces vues, qui seront présentées dans la suite de cette section. Le diagramme de classes est aussi largement utilisé comme base pour les travaux présentés dans cette thèse, ainsi que les diagrammes de séquence et d'états pour illustrer certaines idées. Il existe six autres vues (cas d'utilisation, diagramme d'activité et de collaboration, diagramme d'objets, diagramme de déploiement, et diagramme de paquetage) qui ne sont pas détaillées ici, car elles sont peu exploitées dans le cadre de cette thèse.

2.2.1 Diagramme de classes

Le diagramme de classes est la vue principale pour le langage UML. Il permet de décrire statiquement les classes présentes dans un système et les relations entre ces classes.

Tout d'abord, une *classe* contient plusieurs informations : un nom, des attributs et des méthodes. La Figure 1 montre la notation UML pour une classe LIVRE qui a un attribut titre et trois méthodes `getTitre`, `emprunter` et `retour`. Il existe trois types de *visibilité* pour les classes, les attributs et les méthodes : privé, protégé, public. La visibilité privée restreint l'accès à cet élément aux éléments de la même classe, un élément protégé est accessible aux descendants de la classe et aux éléments d'un même paquetage, et un élément public est accessible à tous les autres éléments du système. La notation UML pour la visibilité est la suivante : - pour privé, # pour partagé et + pour public. Enfin, une classe peut être abstraite ou une simple interface. Une *classe abstraite* peut contenir des attributs et des méthodes, mais le corps de certaines méthodes est vide et elles doivent être définies dans des classes filles. Le nom d'une classe abstraite est noté en italique dans un diagramme de classes. Une *interface* correspond au cas extrême d'une classe abstraite puisque aucun attribut ni aucune méthode n'ont d'implantation. Une classe d'interface sert à spécifier les signatures des méthodes qui seront implantées par toutes ses sous-classes. Le nom d'une interface est précédé par « interface » dans un diagramme de classes. La notation UML pour ces deux types de classe particuliers est donnée Figure 2.

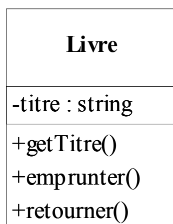


Figure 1 – Une classe Livre

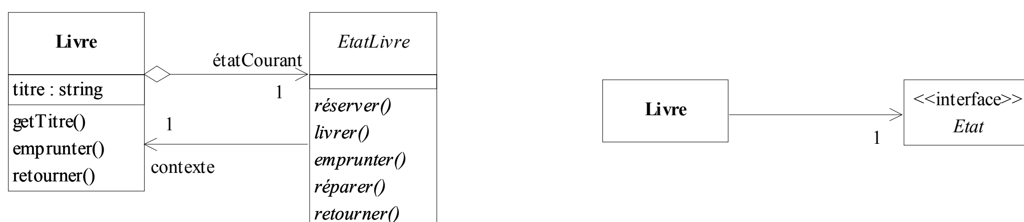


Figure 2 – Classe abstraite et interface

Ensuite il existe des relations entre les classes d’un système. UML permet de décrire trois types de relations : une relation permanente appelée une *association*, une relation temporaire appelée une *dépendance*, et la relation d’*héritage* ou *spécialisation*. Une association peut être renforcée soit par une *agrégation*, soit par une *composition*. La Figure 3 illustre ces cinq types de relation de la classe CLASSB vis-à-vis de CLASSA : dépendance, héritage, association, agrégation, composition. Une association (et une agrégation/composition) peut comporter cinq informations supplémentaires : le rôle de cette association, les rôles de chaque classe pour cette association et les cardinalités de chaque classe pour cette association. La Figure 4 illustre la notation pour ces informations : l’association entre les classes LIVRE et EVTLIVRE joue le rôle des *gestionEvts*, une instance de la classe LIVRE encapsule un ensemble (cardinalité *) d’instances de la classe EVTLIVRE, et cet ensemble joue le rôle de commandes vis-à-vis du livre, enfin une instance de la classe EVTLIVRE est référée par une seule instance (cardinalité 1) de LIVRE qui joue le rôle de sujet.

Le notion d’héritage, présente dans les langages à objets, est un mécanisme important pour l’extension et la réutilisation : quand une classe fille hérite d’une classe mère, elle hérite de tous les comportements de la classe mère, et peut en définir de nouveaux qui lui sont propres. Par ailleurs, l’héritage est un mécanisme d’abstraction. Par exemple, si une classe cliente C utilise les services fournis par une classe fournisseur F qui se spécialise en plusieurs classes (F_1,…, F_N), le client ne sait pas, a priori, quelle classe F_I va

effectivement exécuter le service requis. Le mécanisme qui permet de créer, à l'exécution, une association effective entre deux classes, s'appelle la *liaison dynamique*.

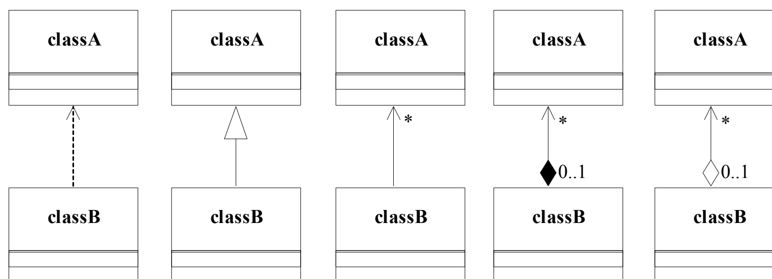


Figure 3 - Cinq relations possibles entre classes

Figure 4 - Les rôles et cardinalités sur une association

2.2.2 Vues dynamiques : diagrammes de séquence et d'états

Les diagrammes de séquence permettent de représenter des collaborations entre objets selon un point de vue temporel, on y met l'accent sur la chronologie des envois de messages. Sur ce diagramme, chaque objet est représenté par un axe vertical. L'ordre d'envoi d'un message est déterminé par sa position sur l'axe vertical du diagramme ; le temps s'écoule "de haut en bas" sur cet axe.

Un exemple de diagramme de séquence est illustré par la Figure 5. Les axes verticaux représentent soit une instance d'un acteur, soit une instance d'une classe. Le diagramme de séquence décrit les échanges de messages entre ces instances, la création dynamique d'objets, et une alternative pour l'échange de messages. Plusieurs constructions graphiques existent pour spécifier l'alternative ou la récursivité dans un diagramme de séquence

Le diagramme d'états permet de représenter un automate d'états finis, sous forme de graphe d'états reliés par des arcs orientés qui décrivent les transitions. Les diagrammes d'états permettent de décrire les changements d'états d'un objet ou d'un composant, en réponse aux interactions avec d'autres objets/composants ou avec des acteurs. Un état peut être simple, ou composé lui-même de sous-états et de transitions entre ces états (on parle alors de super-état). On distingue un état initial, et il peut y avoir plusieurs états terminaux.

Une transition est atomique et représente donc le passage supposé instantané d'un état vers un autre. Trois éléments peuvent étiqueter une transition : une *garde*, un *évènement* et une *action*. La transition est déclenchée par l'évènement, la garde est une expression

booléenne qui conditionne le déclenchement de la transition si elle existe. Enfin, il est possible d'associer une action à l'évènement, ce qui, en pratique, revient à dire que l'on fait un traitement dans le système (appel de méthode, changement de valeur d'un attribut…).

La Figure 6 illustre un exemple de diagramme d'états. L'état `ordered` est l'état initial. Si le nombre de réservations est nul et que l'opération `deliver` est exécutée, l'état courant passe à `available`. Les états `available` et `reserved` sont regroupés dans un état composite `InLibrary`.

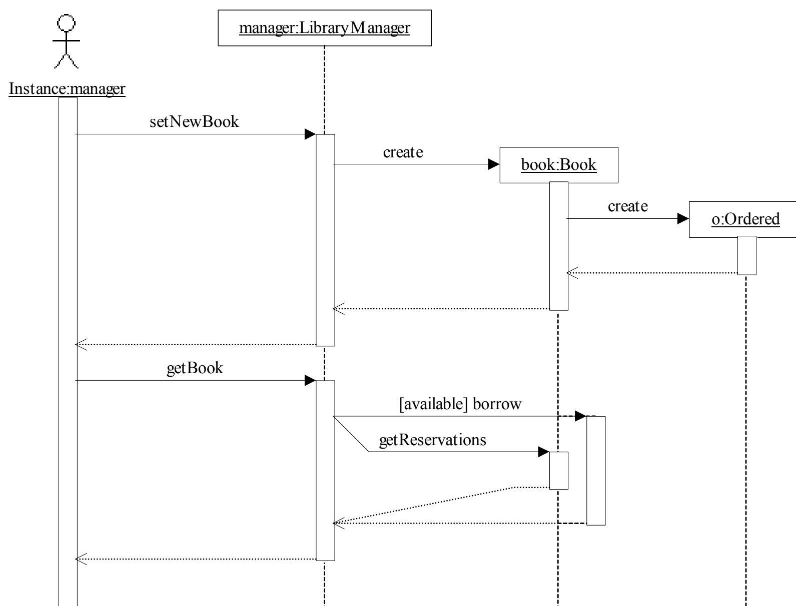


Figure 5 – Exemple de diagramme de séquence

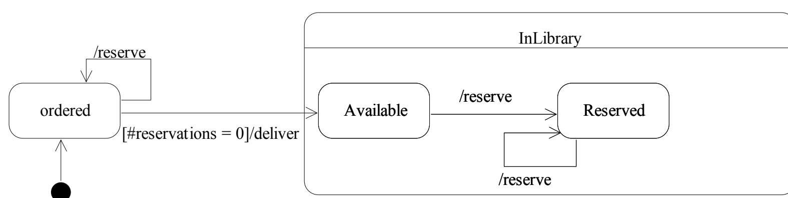


Figure 6 – Exemple de diagramme d'états

2.3 Méthodologie pour une conception testable

Le succès de l'approche objet tient beaucoup à la notion de composant réutilisable. La réutilisation de composants logiciels devient un enjeu crucial tant pour éviter de réécrire inutilement du logiciel, que pour réduire les coûts de développement. On cherche alors à

développer la notion de composant objet « sur étagère », et il faut pour cela pouvoir mesurer la confiance que l'on peut avoir en un composant.

Nous introduisons ici une méthode pour la conception de composants fiables fondée sur le test. Cette méthode considère un composant comme l'agrégation d'une implantation, d'une spécification et d'un ensemble de cas de test. L'idée consiste alors à considérer qu'on peut avoir confiance dans un composant si ces trois aspects sont cohérents l'un vis-à-vis de l'autre. Pour vérifier cette cohérence, nous rendons la spécification partiellement exécutable grâce à des contrats embarqués dans le composant, et nous utilisons l'analyse de mutation pour vérifier la cohérence entre les trois aspects.

Nous commençons par présenter les principes de la conception par contrat, et comment cette méthode peut être appliquée avec UML. Nous détaillons ensuite la méthode proposée par Triskell pour concevoir des composants fiables appelés *composants autotestables*.

2.3.1 La conception par contrat

La notion de contrat a été définie pour capturer les droits et obligations de chaque classe. L'expérience montre que le simple fait d'exprimer ces contrats de manière non ambiguë est une approche de conception valide [Jézéquel"97]. Bertrand Meyer a nommé cette approche de la construction de logiciel la conception par contrats (Design by Contract) [Meyer"92b]. Cette approche repose sur les théories des fonctions partielles et de la logique des prédicats. Elle offre aussi une méthodologie pour construire des systèmes modulaires, extensibles et réutilisables tout en prenant en compte la robustesse du système.

La conception par contrats est à l'opposé de la programmation défensive qui recommande de mettre autant d'assertions que possible dans le code [Liskov"86]. La programmation défensive complique la localisation d'une faute parmi les différents modules et augmente la complexité du logiciel ce qui peut entraîner des erreurs supplémentaires.

La conception par contrats conduit les développeurs à bien séparer les responsabilités de l'appelant d'une méthode (le client) et de l'implantation de cette méthode. Ces contrats sont définis par des expressions booléennes appelées *précondition* et *postcondition* pour les routines. On peut aussi définir un invariant de classe pour définir les propriétés globales de l'occurrence d'une classe qui doivent être vérifiées par toutes les routines. Un contrat définit les droits et obligations de chaque partie : le client doit appeler une méthode uniquement en respectant la précondition de la méthode et l'invariant de la classe à laquelle elle appartient. En échange, la méthode appelée garantit que la propriété définie par la postcondition ainsi que l'invariant de classe sont vérifiés.

Si un contrat n'est pas respecté, une erreur a été détectée. La violation d'une précondition signifie que le client n'a pas respecté le contrat : dans ce cas, la méthode

appelée ne peut pas respecter sa part du contrat mais peut signaler la faute en soulevant une exception. La violation d'une postcondition signale une erreur dans l'implantation de la méthode appelée qui ne peut pas respecter sa part du contrat.

La conception par contrats est intégrée au système de typage des langages orientés objet grâce à la notion de sous-contrat fournie par le mécanisme d'héritage. En effet, la liaison dynamique permet à une routine de passer un contrat avec une redéfinition pour son implantation. La redéfinition est alors une transformation qui préserve la sémantique puisque la nouvelle définition doit au moins remplir le contrat de la méthode originale, et éventuellement en faire plus (c'est à dire qu'elle peut accepter des appels qui n'étaient pas acceptés par l'originale, et fournir un «meilleur » résultat). C'est pourquoi une précondition dans une sous-classe peut-être affaiblie (accepter plus), et une postcondition renforcée (faire plus).

Dans la suite de cette thèse, nous nous intéressons à l'apport qualitatif de la conception par contrat. Nous étudions comment et à quel moment dans le développement les contrats permettent d'améliorer la qualité d'un composant. Par exemple, de nombreux travaux soulignent l'importance des contrats pour la détection et la localisation des erreurs. Dans le chapitre 3, nous proposons un modèle pour évaluer l'apport des contrats pour le diagnostic (phase de localisation des erreurs), en fonction de leur qualité et de leur répartition dans le système. Nous étudions aussi l'apport des contrats pour la détection d'erreur en proposant une mesure de la robustesse d'un système orienté-objet.

2.3.2 Les contrats et UML : le langage OCL

Le langage OCL (Object Constraint Language, [OMG"97]) est un langage formel pour exprimer des contraintes sur des diagrammes UML. Il permet de décrire un invariant pour une classe, et des pré et post conditions pour des méthodes. OCL permet donc de décrire des contrats sur un modèle UML, et sera utilisé dans la suite pour illustrer certains points lors de l'étude de la qualité des contrats pour un composant ou un assemblage de composants.

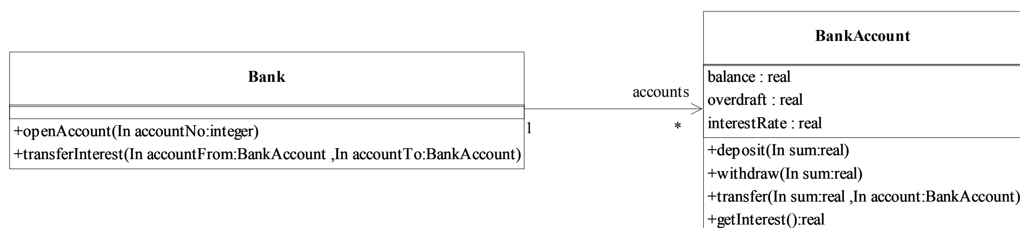


Figure 7 – Diagramme de classes pour une banque

Dans la suite de cette section, nous illustrons les différents types de contraintes OCL, ainsi que leur syntaxe, sur l'exemple de la Figure 7. Un invariant peut être exprimé pour

chacune des classes BANK et BANKACCOUNT. Une contrainte de type invariant est désignée par le mot-clé *inv*, et le type sur lequel la contrainte doit être appliquée est désigné par le mot-clé *context*. Par exemple, un invariant pour la classe BANKACCOUNT, spécifiant que la valeur de `balance` doit supérieure ou égale à la valeur de `overdraft`, pour tout instance de BANKACCOUNT, s'écrit de la manière suivante :

```
context bankAccount inv:
 self.balance >= self.overdraft
```

OCL permet de naviguer les associations à partir d'un objet particulier, le nom du rôle désigne l'ensemble des objets de l'autre côté de l'association. Par exemple, un invariant qui contraint toute instance de BANK à avoir au moins un compte s'écrit de la manière suivante :

```
context Bank inv :
 self.accounts->size>0
```

Les contraintes de type pré et post condition sont désignées par les mots-clé *pre* et *post* dans OCL. Dans les postconditions, OCL permet d'accéder à la valeur de l'état avant l'opération grâce à l'opérateur *@pre*. Une précondtion pour la méthode `deposit` de la classe BANKACCOUNT consiste à vérifier que le montant passé en paramètre est positif. Une postcondition vérifie que la valeur de `balance` après l'opération est égale à la valeur avant l'opération plus le montant passé en paramètre. Ces contraintes s'expriment de la manière suivante :

```
context BankAccount::deposit(sum:real):void
  pre sum > 0
  post self.balance = self.balance@pre + sum
```

2.3.3 Les composants autotestables

La solution envisagée dans le cadre du projet Triskell, pour la conception de composants de confiance, consiste à considérer chaque composant comme un tout possédant une spécification, une implantation et un ensemble de cas de test mémorisés et pouvant être activés (Figure 8). Des composants conçus avec cette méthode sont dits *autotestables* [Jézéquel"01]. Cette approche conceptuelle est généralisable à des échelles plus grandes que le composant : des assemblages de composants autotestables sont autotestables. A tous les niveaux de complexité, les composants ont la possibilité d'exécuter les cas de test qui leur sont associés.

Avec la conception par contrat [Meyer"92a; Jézéquel"99], des contrats exécutables sont dérivés de la spécification. De cette manière, les trois aspects du composant autotestable sont exécutables, et peuvent être confrontés l'un à l'autre et améliorés de manière incrémentale pour augmenter la confiance dans le composant.

Un point important lors de la conception de composants logiciels destinés à être utilisés dans différents environnements, est d'être capable de leur associer une mesure de confiance. Le terme anglais « trustability » [Howden"70], désigne cette propriété qu'il est difficile d'estimer directement : on ne peut l'évaluer qu'en analysant certains facteurs qui

influencent cette propriété. Dans le cadre de la méthode des composants autotestables, nous considérons la qualité de l'ensemble des cas de test associé au composant comme le facteur principal pour l'évaluation de la confiance. En effet, un ensemble efficace de cas de test doit permettre d'améliorer l'implémentation et les contrats, et d'obtenir ainsi une forte cohérence entre ces aspects, ce qui permet d'avoir confiance – de manière indirecte – dans le composant.

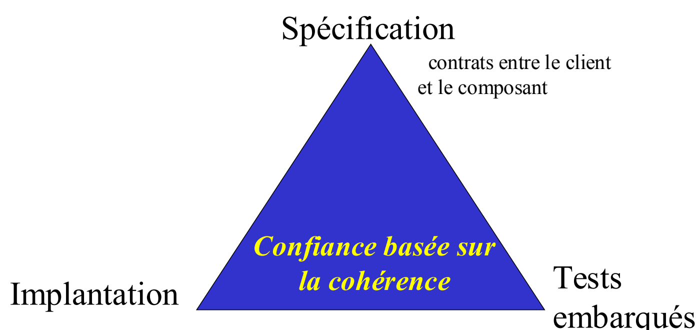


Figure 8 – Composant autotestable – la vue en triangle

Méthodologiquement, on souhaite donc renforcer la cohérence entre les trois facettes d'un composant (spécification/implantation/cas de test) en les confrontant l'un à l'autre. Une fois qu'un ensemble de cas de test efficaces est disponible, la correction des erreurs détectées améliore la facette implantation. Pour l'amélioration de la partie spécification exprimée à l'aide de contrats, l'idée est la suivante : si les contrats sont complets, ils devraient être violés quand une erreur est présente dans l'implantation, et qu'un cas de test exécute la partie erronée, et déclenche l'erreur. De plus si les contrats sont corrects, ils ne devraient être violés que dans ces cas-là. Donc, si on exécute un ensemble de cas de test efficaces sur une version du programme dans laquelle on a placé une erreur, et qu'aucun contrat n'est violé, alors une faiblesse dans les contrats a été détectée. Les contrats qui auraient dû détecter l'erreur doivent alors être complétés. D'autre part, lorsque l'ensemble de cas de test est exécuté sur l'implantation et qu'un contrat est violé, si aucune erreur n'est trouvée dans le programme, il faut vérifier que ce n'est pas dû à une erreur dans un contrat.

Un critère pour évaluer l'efficacité d'un ensemble de cas de test est donc nécessaire à l'application de cette méthodologie. Nous aurions pu, dans un premier temps, recourir à un critère de couverture (type couverture de code), mais cela ne reflèterait que de manière très lointaine la capacité des cas de test à révéler des erreurs [Voas"92a]. Dans cette optique, la technique –certes plus lourde – qui nous a paru la plus proche de l'activité effective du test, est l'analyse de mutation, à condition de l'adapter aux langages OO et de permettre le passage à l'échelle (test d'un assemblage de composants).

L'analyse de mutation est la technique centrale qui a été choisie pour tout ce processus d'amélioration, et elle présentée en détail dans la section 2.1.3. Bien que ce

soit avant tout une technique pour améliorer et valider un ensemble de cas de test, nous venons d'évoquer comment elle pourrait être utilisée pour valider les contrats.

La conception de composants autotestables permet aussi de résoudre des problèmes de test dans le cas de l'évolution et de la réutilisation de composants. En effet, comme les cas de test sont une partie intégrante du composant, il porte la capacité de se tester (autotest), et ses cas de test peuvent être appelés depuis le système. Trois cas de figures peuvent être pris en compte pour un composant C dans un système S :

- *modification de l'implémentation de C* : l'autotest permet de tester la nouvelle implémentation, la spécification ne change pas,
- *ajout de nouvelles fonctionnalités à C* : l'autotest doit être lancé pour assurer le test de non-régression, puis l'ensemble de cas de test et la spécification doivent être complétés pour prendre en compte les nouvelles fonctionnalités. Enfin, tous les composants clients de C doivent s'autotester pour garantir la non-régression.
- *réutilisation du composant dans un système S* : lors de l'ajout d'un composant C dans un système, l'autotest de C doit être lancé depuis le système pour vérifier sa bonne intégration (héritage et clientèle).

L'approche développée apporte donc un support pour aider à résoudre le problème du test d'intégration, du test de non-régression, de l'évolution des composants et de leur réutilisation. Elle a pour principale limitation le fait qu'elle ne permet pas le test fonctionnel du système, et qu'elle ne modélise pas explicitement les aspects comportementaux, dynamiques du système. C'est ce dernier aspect, qu'il faudrait prendre en compte de manière explicite (aspects de communication entre objets) en proposant des modèles et des méthodes de test spécifiques.

2.3.4 Conclusion

Au cours de cette section, nous avons présenté la conception par contrats. Cette technique est utilisée pour rendre une partie de la spécification exécutable, ce qui permet de confronter l'implantation et la spécification grâce au test dynamique. Le test permet donc d'améliorer la cohérence entre ces deux aspects d'un composant et ainsi d'augmenter la confiance dans ce composant. Nous avons présenté une méthode proposée par l'équipe Triskell, et fondée sur la cohérence entre les cas de test, l'implantation et la spécification d'un composant, pour assurer un certain niveau de confiance dans le composant.

Les trois sections suivantes détaillent l'état de l'art spécifique aux trois volets de cette thèse.

2.4 La génération automatique de cas de test

Nous détaillons ici les principaux travaux sur la génération automatique de cas de test, travaux qui sont en lien avec le chapitre 3 dans lequel nous exposons notre contribution à la génération de test pour des composants.

La génération automatique de données de test est un problème crucial pour la construction de logiciels fiables [Dustin"99; Fewster"99]. L'efficacité de la phase de test dépend en grande partie de la pertinence de données utilisées. Les techniques pour la génération automatique sont de deux types : fonctionnelle ou structurelle.

2.4.1 Techniques fonctionnelles

Les techniques de génération fonctionnelle se fondent sur une spécification du comportement du programme. Il existe un ensemble de travaux sur la génération automatique de cas de test à partir d'une spécification formelle d'un système. Ces travaux s'appuient sur la méthode présentée dans [Dick"93] qui consiste à générer un automate d'états finis à partir de la spécification formelle, puis à sélectionner des cas de test comme des chemins dans l'automate. Dans [Legeard"02], les auteurs utilisent une spécification B ou Z du programme à tester. Ces spécifications sont ensuite traduites au format BZ-Prolog proposé par les auteurs, chaque opération est alors décrite par son nom, les noms et types des paramètres d'entrée et de sortie et les pré et post conditions. À partir d'une telle description, Legeard et al. décrivent une technique pour générer automatiquement des données de test aux limites, ainsi que les préambule et postambule nécessaires à l'exécution du cas de test. Dans [Murray"98], Murray et al. utilisent une spécification Object-Z d'une classe pour générer un automate et générer des cas de test unitaires pour la classe. Sur la même idée, les auteurs de [Grieskamp"02], dérivent un automate d'états fini à partir d'une machine à états abstraite pour générer des cas de test.

Un autre exemple de génération automatique de cas de test fondée sur une spécification formelle est décrit dans [Marinov"01]. Ici, les auteurs utilisent Alloy, un langage qui permet de spécifier le comportement de méthodes Java. D'autres approches se basent sur l'expression des exigences pour la génération de tests fonctionnels [Tahat"01] ou sur la description de scénarios abstraits [Tsai"01; Nebut"02].

Dans [Pickin"02] les auteurs décrivent une solution fondée sur une spécification formelle du système. Cette approche prend en entrée un diagramme de classes, les machines à états décrivant le comportement de chaque classe, un déploiement initial du système et un diagramme de séquence décrivant un objectif de test. A partir de cette spécification, un simulateur intégré à l'atelier de génie logiciel UMLAUT [Ho"99] peut générer un graphe étiqueté décrivant le comportement global du système, et un graphe pour l'objectif de test. Ces deux graphes sont donnés en entrée de l'outil TGV [Jéron"98] qui génère un cas de test exécutable sur l'implantation du système.

D'autres travaux sur la génération automatique de test fondée sur les modèles UML ont été réalisés dans le cadre du projet européen AGEDIS [AGEDIS"00]. Ici, les techniques étudiées utilisent le diagramme de classes, des diagrammes d'objets pour spécifier des situations initiales d'utilisation du système, et les diagrammes d'états pour spécifier le comportement dynamique des objets. A partir de ces différents éléments de spécification une méthode ainsi que des outils pour la génération automatique de cas de test ont été proposés.

2.4.2 Techniques structurelles

Parmi les techniques structurelles, on distingue la génération statique de la génération dynamique. La *génération statique* consiste à générer des données de test sans exécuter le programme sous test. Des outils de génération statique sont fondés sur la résolution de contraintes sur les données en entrée. Un exemple d'une telle technique est l'exécution symbolique de programme [Clarke"76], qui consiste à donner une valeur symbolique à chacune des variables du programme. L'exécution statique du programme permet ensuite d'obtenir une expression représentant une contrainte sur les variables, nécessaire pour atteindre un point du programme. Dans [Lugato"02], les auteurs utilisent AGATHA, un outil fondé sur l'exécution symbolique, pour générer des données de test à partir d'une spécification avec UML.

La *génération dynamique de cas de test* initialement proposée dans [Miller"76] consiste à instrumenter le code du programme sous test pour collecter des informations sur le programme au cours de l'exécution d'un cas de test. Ces informations sont utilisées pour évaluer la qualité du cas de test par rapport à un critère donné. Les données de test sont ensuite modifiées pour tenter d'améliorer le cas de test afin qu'il satisfasse entièrement le critère choisi. De nombreux travaux sur la génération dynamique de cas test sont inspirés des travaux de Korel qui ramène ce problème à un problème de minimisation de fonction [Korel"90; Korel"92; Korel"96]. L'idée est la suivante : un objectif de test peut s'exprimer sous la forme d'une fonction sur les données en entrée, trouver une donnée de test pertinente consiste alors à trouver une donnée qui minimise la fonction. Korel utilise la descente de gradient pour résoudre la minimisation de fonction. Un grand nombre de chercheurs ont utilisé un algorithme génétique pour résoudre ce problème [Xanthakis"92; Schultz"93; Boden"96; Jones"96; Jones"98; Tracey"98; Pargas"99; Tracey"00; Michael"01; Wegener"01]. Ces travaux, que nous détaillons dans la suite, diffèrent par la modélisation des prédicats comme fonction des variables d'entrée.

Dans [Michael"01], Michael et al. présentent leur outil GADGET (Genetic Algorithm Data GEneration Tool) qui génère des données de test scalaire. Les individus sont modélisés comme une chaîne de bits correspondant à la représentation binaire de la donnée de test et la fonction d'utilité correspond à la fonction qu'on cherche à minimiser. L'article détaille la modélisation du problème et donne des résultats expérimentaux avec plusieurs petits programmes (30 lignes de code) et un programme plus important (2000

lignes de code). Au cours de ces expériences, les auteurs ont utilisé deux types d'algorithme génétique pour la génération de données de test. Les résultats sont comparés à la génération aléatoire, et à l'utilisation de la descente de gradient pour la minimisation de fonction. La génération purement aléatoire est beaucoup moins efficace que les autres techniques dès que la taille du programme sous test devient importante, ce qui justifie l'effort supplémentaire nécessaire à l'application des méthodes plus sophistiquées. Parmi les trois algorithmes de génération, l'algorithme génétique classique est le plus efficace dans une large majorité des cas.

Tracey et al. [Tracey"00] utilisent un algorithme génétique pour la génération de données de test pour les conditions d'exception. Ils présentent le calcul de la fonction d'utilité et détaillent un exemple. Ils donnent ensuite des résultats expérimentaux pour des programmes écrits en Ada95. Le plus gros exemple est un système de contrôle pour un moteur d'avion, pour lequel l'algorithme est capable de générer des données de test qui soulèvent chaque exception possible.

Jones et al. [Jones"96; Jones"98] donnent différents calculs possibles pour la fonction d'utilité. Pour tous ces modèles, les prédicats sont vus comme une fonction des données en entrée. Il faut ensuite trouver des données de test qui minimisent ces fonctions. Dans [Jones"98], les mêmes auteurs proposent d'utiliser aussi le score de mutation comme fonction d'utilité.

Pargas et al. [Pargas"99] utilisent aussi un algorithme génétique pour la génération dynamique de données de test. Ici, la fonction d'utilité est fondée sur la couverture du graphe de dépendance de contrôle (GDC). Les auteurs ont développé un prototype appelé TGen. L'outil génère une donnée de test pour un objectif de test. Cet objectif correspond à un chemin du GDC que l'on veut couvrir. En exécutant le cas de test sur une version instrumentée du programme sous test, TGen repère les nœuds couverts par le cas de test. La valeur d'utilité du cas de test est évaluée par le nombre de nœuds en commun entre le chemin couvert et le chemin correspondant à l'objectif de test. L'article décrit précisément le processus de génération fondé sur l'algorithme génétique, puis plusieurs expériences sur 6 petits programmes (entre 32 et 82 lignes de code). Les résultats de l'algorithme génétique sont comparés aux résultats obtenus avec un générateur aléatoire.

Tous les travaux sur les algorithmes génétiques présentés ici offrent des solutions qu'il est difficile d'adapter dans un cadre orienté objet. Tout d'abord, ils ont en commun de générer des données de type simple, ce qui leur permet de modéliser la solution sous forme d'une chaîne binaire. Or, dans le cas de programmes orientés objet, nous devons générer des objets complexes qu'il est difficile de modéliser sous forme binaire. Par ailleurs, ces travaux exécutent l'algorithme génétique pour chaque objectif de test, c'est-à-dire qu'ils ne génèrent qu'une donnée de test à chaque exécution. Dans le cadre de la méthodologie nous voulons que l'algorithme optimise un ensemble de cas de test pour un

composant, c'est-à-dire qu'il génère toutes les données nécessaires pour tester correctement le composant en une seule exécution.

Dans le chapitre 3, nous étudions donc un algorithme génétique pour l'optimisation et la génération automatique de cas de test, et proposons d'adapter cet algorithme pour résoudre les problèmes soulevés ci-dessus. Ces adaptations donnent lieu à un nouvel algorithme semi-aléatoire que nous baptisons algorithme bactériologique.

2.5 Conception par contrat et test

Dans cette thèse nous insistons sur l'intérêt des contrats tant pour la qualité d'un composant que pour assurer une qualité de l'assemblage (c'est le thème du chapitre 4). Pour avoir une idée de l'impact des contrats sur cette qualité nous proposons une étude de l'apport d'une conception par contrat [Meyer"92b] pour la robustesse et la diagnosabilité d'un système orienté objet. Au cours de ces travaux nous avons étudié des méthodes pour l'écriture des contrats, essayé de distinguer différentes qualités pour ces contrats en vue d'une classification, et enfin nous nous sommes intéressés aux contrats pour aider le test de logiciel. Cette section dresse l'état de l'art sur différents points : les méthodes pour la conception par contrat, la classification des contrats et enfin, les contrats et plus généralement, les assertions, pour le test de logiciel.

Plusieurs articles traitent du problème de la rédaction des contrats, en se concentrant sur différents aspects de cette rédaction pour proposer une méthode ou une classification. Le but de tous ces travaux est de proposer une aide pour écrire des contrats de manière efficace.

Les auteurs de [Collet"96] proposent une classification des contrats sur deux critères, d'ordre plus méthodologique que syntaxique. Le premier critère prend en compte les entités utilisées par les contrats, pour aider à placer les contrats de manière efficace. Le second critère se réfère à l'intention du programmeur, c'est-à-dire à la granularité jugée nécessaire par le programmeur. David Rosenblum propose aussi une classification empirique des contrats dans [Rosenblum"95]. Les critères choisis dans cet article sont spécifiques au langage C, et le but est de classifier les types d'erreur détectée par les contrats, plus que de fournir une méthode de rédaction de ces contrats.

D'autres équipes se sont intéressées à des règles minimales pour la rédaction de contrats efficaces. McKim [McKim"95] établit une liste minimale de points qui doivent être vérifiés lors d'une conception par contrats. Cette liste est complétée dans [Mitchell"99], à partir de l'observation de l'application du design pattern Observateur [Gamma"95], les auteurs de cette étude repèrent des problèmes qui apparaissent lors de la rédaction de contrats pour un ensemble de classes plutôt que pour une classe isolée. Le but de ces travaux est d'améliorer les performances, et l'extensibilité des composants.

Concernant les méthodes pour utiliser les contrats pour le développement d'un logiciel, Nordby, Blom et Brunstrom présentent une méthode de développement de logiciel fondée sur les contrats dans [Nordby"02]. Ils distinguent des contrats forts et des contrats faibles, qui correspondent à deux niveaux de granularité : les contrats forts établissent des obligations que le client doit vérifier pour utiliser la routine, la post-condition se contente alors d'exprimer le résultat dans le cas où la routine est utilisée correctement. Pour un contrat faible, la pré-condition sera typiquement à vrai, relâchant ainsi les contraintes sur le client. La post-condition devra alors exprimer le résultat attendu dans le cas d'une utilisation correcte et d'une utilisation incorrecte (message d'erreur, exception…). Les deux types de contrat sont utilisés à des moments différents du développement : les contrats forts pour le développement des unités et la découverte d'erreurs internes, les contrats sont ensuite affaiblis au moment de l'intégration pour tester la robustesse du composant. Dans [Nordby"02], les auteurs présentent une manière sûre d'affaiblir des contrats, et donnent des résultats d'application de cette méthode dans un cadre industriel. Les contrats forts ont permis de détecter rapidement les erreurs au cours du développement de deux composants. Les contrats ont ensuite été affaiblis, et cette opération pour passer des contrats forts aux contrats faibles n'a pas introduit d'erreur.

De manière générale, la conception contrats peut être vue comme une méthode particulière pour placer des assertions dans un programme. Lorsque ces assertions sont exécutables, elles fournissent un moyen puissant de détection d'erreurs. Cependant, peu de travaux se sont intéressés aux assertions pour tester un programme.

Les assertions placées dans le code d'un programme peuvent être utilisées comme fonction d'oracle pour le test. Elles vérifient la cohérence de l'état du logiciel à certains points précis. La violation d'une assertion signifie donc que le programme est passé dans un état erroné, et qu'une opération a rendu un résultat en contradiction avec sa spécification. Dans [Fenkam"02] les auteurs étudient la dérivation d'assertions pour l'oracle de test de composants CORBA à partir d'une spécification formelle du programme. Les auteurs de [Cheon"02] utilisent les post-conditions de méthodes et les invariants de classe exprimés avec JML (Java Modeling Language) pour générer des oracles pour le test de classes Java. JML est un langage pour écrire des pré et post conditions et des invariants pour des programmes écrits en Java. Les oracles sont automatiquement générés dans une classe au format JUnit. Edwards propose aussi d'utiliser les assertions embarquées dans le logiciel comme oracle pour le test de composants logiciels [Edwards"01], sans cependant étudier leur efficacité ou leur complétude.

Un autre aspect du test pour lequel les assertions s'avèrent utiles, est la testabilité. La testabilité est un facteur de qualité qui exprime la difficulté pour tester un programme (tant du point de vue de l'effort de test que de la difficulté de détecter les erreurs). Un des points essentiels, pour découvrir des erreurs au cours du test, est de pouvoir observer le

comportement du programme à différents moments de l'exécution. En effet, si on observe une erreur au milieu de l'exécution, on sait que l'erreur se trouve dans la partie du programme exécutée avant ce moment, alors que si on n'observe l'erreur qu'à la fin, la localisation de l'erreur est plus difficile. Voas a beaucoup étudié ces problèmes d'observabilité, et propose une méthode pour placer des assertions de manière efficace pour améliorer la testabilité du programme [Voas"99].

Enfin, Korel [Korel"96] utilise les assertions pour générer des données de test automatiquement. Son idée est de générer des données en entrée qui violent des assertions dans le programme. S'il trouve de telles données, qui sont conformes à la spécification des données en entrée du programme sous test, elles ont forcément détecté une erreur qui peut être de l'un de trois types suivants : une erreur dans le programme, une erreur dans une assertion, une pré-condition trop faible. Pour générer de telles données, il ramène le problème à celui de la génération de données de test qui atteignent un point particulier du programme et le résout comme dans [Korel"92].

En conclusion, il existe d'une part des travaux sur des aspects précis de la rédaction de contrats pour un programme, et d'autre part des travaux sur l'utilisation des assertions exécutables pour aider le test. Cependant, il semble qu'aucune étude ne se soit consacrée sur l'impact des contrats sur le test. Dans la suite de cette thèse (chapitre 4), nous proposons une évaluation de l'apport des contrats pour la détection et la localisation d'erreurs.

2.6 Testabilité et mesures pour les logiciels OO

La testabilité, que nous étudions pour un assemblage de composants au chapitre 5, est à la frontière de deux domaines. D'une part, elle est liée aux problèmes de test en évaluant l'effort requis pour examiner un logiciel. D'autre part, la testabilité est une mesure. Ainsi, cette section présente les travaux existants sur l'aspect mesure de la testabilité, et introduit certains travaux plus généraux sur la mesure des logiciels OO.

La mesure de testabilité est un critère de qualité qui évalue, à partir de caractéristiques structurelles d'un programme, l'effort nécessaire pour le tester. Cette mesure peut être définie ainsi :

Testabilité. *La testabilité est un facteur de qualité, défini comme la facilité à tester un logiciel. Cette facilité est à la fois une propriété intrinsèque du schéma de conception (et donc une caractéristique propre au produit étudié) et une propriété relative à la stratégie de test adoptée pour vérifier un critère de test particulier. La testabilité se dérive en trois attributs :*

- *le coût global de test : le coût global pour obtenir des cas de test vérifiant un critère de test donné*

- *la contrôlabilité : la facilité pour générer des données de test efficaces (propriété intrinsèque au logiciel sous test)*
- *l'observabilité : la facilité avec laquelle il est possible de vérifier la pertinence des résultats obtenus après l'exécution des cas de test*

De manière générale, la testabilité vise deux choses : un but technique pour un concepteur et un but de gestion de projet. Pour le concepteur du logiciel c'est un indicateur qui permet, dés les premières phases de la conception, d'identifier les systèmes qui peuvent devenir difficiles à tester. Pour le chef de projet, cette mesure permet de trouver un compromis en terme de coûts parmi différentes solutions fondées sur diverses conceptions et techniques de test.

Quel que soit le facteur de qualité observé, voici deux exemples d'informations que le concepteur peut espérer obtenir :

- localisation des points délicats dans la conception où le facteur peut avoir une valeur faible, et où l'architecture doit être améliorée
- détection des changements inadaptés (raffinements, évolutions) qui peuvent mener à une variation non désirée de ce facteur

L'objectif du test de logiciel n'est pas seulement de couvrir et exécuter chaque partie du logiciel mais aussi de révéler des fautes cachées. De ce point de vue, les notions de contrôlabilité et d'observabilité d'un composant logiciel, introduites par Freedman [Freedman"91], sont complémentaires du coût de test. Par exemple, la contrôlabilité est relative au rapport entre la taille du domaine des données de sortie et celle du domaine des données en entrée du logiciel. Cependant, son approche ne prend pas en compte la difficulté qu'il y a à exécuter un composant et à propager l'erreur jusqu'à la sortie du logiciel. Cette limite a été analysée en détails par Voas [Voas"92b; Voas"95] au niveau du code du composant. Plusieurs mesures ont été proposées pour évaluer la perte d'information dés la conception, telles que le Domain/Range Ratio de Voas [Voas"93] pour les programmes impératifs ou la mesure de contrôlabilité/observabilité pour les architectures flots de données [Le Traon"97; Le Traon"00b]. Weide et al. [Weide"96] ont défini l'observabilité et la contrôlabilité des types de données abstraits pour une meilleure réutilisation des composants logiciels. Quant à la mesure de la contrôlabilité et l'observabilité dans les programmes OO, aucune approche satisfaisante n'a été proposée.

En ce qui concerne l'étude de testabilité présentée au chapitre 5 de cette thèse, nous nous concentrons sur la mesure du coût global de test.

Coût global de test. *Ce facteur évalue le coût nécessaire à la vérification d'un critère de test donné. Cette mesure est relative à la taille de l'ensemble de cas de test, à la difficulté de trouver des données de test pour vérifier le critère ainsi qu'à la difficulté d'évaluer la valeur d'oracle.*

A propos des mesures du coût global de test, Bieman et Schultz [Bieman"89] ont compté combien de cas de test sont nécessaires pour vérifier le critère de couverture de tous les chemins définition-utilisation[Rapps"85]. Il existe de nombreuses mesures pour les architectures OO (e.g. [Chidamber"94; Shepperd"98]), mais la testabilité n'est jamais mesurée directement. Cependant, elle peut être évaluée à partir de la mesure de couplage sous l'hypothèse qu'un fort couplage dans une architecture OO entraîne une diminution de la testabilité. Le couplage mesure la force de la relation entre deux modules, i.e. des classes dans le cas des modèles orientés objet. Un grand nombre de mesures de couplage ont été proposées, chacune correspondant à un type de relations entre les classes. Dans [Briand"99], les auteurs proposent un classement des mesures de couplage pour des éléments de conception UML précis. La mesure de couplage entre objets (CBO) [Shyam"94; Briand"99] correspond à un ensemble de classes qui emploient chacune l'autre. Dans le diagramme de classes UML, on dit qu'une classe A emploie une autre classe B s'il existe une association ou une dépendance entre ces classes. La mesure CBO est discutée en termes de testabilité dans [Binder"94], et des critères de test pour ce type de relations parmi les classes sont proposés dans [Alexander"00]. Ces travaux se concentrent sur chaque chemin indépendamment et visent le comptage/couverture.

Dans le chapitre 5, nous nous concentrons sur des chemins particuliers dans le diagramme de classes, ainsi qu'à la complexité des interactions décrites par ces chemins (due au polymorphisme et à la liaison dynamique). À notre connaissance, la contribution précise à la testabilité de chaque dépendance participant au couplage n'a jamais été étudiée, particulièrement dans le cas de logiciels conçus en utilisant UML. Le but de l'analyse de testabilité proposée est donc plus d'indiquer les rôles des liens participants au couplage que de limiter le nombre de tels liens.

2.7 Conclusion

Le paradigme objets, ainsi que les constructions spécifiques aux divers langages orientés objets ont entraîné la nécessité d'une adaptation de certains aspects du test de logiciel. Un grand nombre de travaux sur les test de programmes objets se fondent aujourd'hui sur le langage UML comme langage de description. Au cours de cette thèse, nous n'utilisons pas de propriétés particulières des diagrammes UML pour le test, mais utilisons la notation pour illustrer nos différents travaux.

Nous nous intéressons à trois aspects particuliers du tests pour la programmation objets. Tout d'abord, nous étudions la génération automatique de cas de test efficaces (section 2.4) pour le test unitaire de classe ou de composant. Ensuite, nous nous intéressons au lien entre les assertions et le test de systèmes OO (section 2.5). Pour cela, nous mesurons l'impact de la conception par contrats, comme une méthode de placements d'assertions dans une architecture OO. Cet impact est évalué sur deux facteurs de qualité d'un système : la robustesse et la diagnosabilité. Ces deux facteurs évaluent la capacité de détection et de localisation d'erreurs respectivement, et sont donc

directement liés à l'efficacité du test. Enfin, nous étudions la complexité des interactions entre composants, et proposons une mesure de la testabilité (section 2.6) d'un assemblage de composants.

3
Génération automatique de cas de test pour un composant

Le test de logiciel est un moyen efficace pour estimer la confiance qu'il est possible d'avoir dans un composant logiciel. Dans le cadre de la méthode des composants autotestables, la mesure de confiance est fondée sur l'évaluation de la cohérence entre l'implantation, la spécification (embarquée dans le composant sous forme de contrats exécutables) et les cas de test. Cette cohérence étant évaluée grâce au test du composant, la qualité des cas de test est alors le facteur principal pour évaluer la confiance. Comme nous l'avons vu section 2.3.3, nous utilisons l'analyse de mutation pour qualifier un ensemble de cas de test, et ce chapitre a pour but l'étude d'algorithmes évolutionnistes pour l'optimisation automatique du score de mutation d'ensembles de cas de test.

Alors que la génération d'un ensemble de cas de test initiaux est simple, améliorer la qualité de cet ensemble demande un effort beaucoup plus grand. En effet, notre expérience montre que les cas de test fournis par le testeur détectent souvent entre 50 et 70% des mutants [Deveaux"99a], ce qui correspond à peu près au test des cas d'utilisation normaux du programme sous test. Par contre, améliorer ce score pour couvrir plus de 90% des mutants est très coûteux tant en effort de génération qu'en temps. Le travail décrit dans ce chapitre se concentre sur l'automatisation de cette phase d'amélioration d'un ensemble de cas de test initiaux.

Le problème de l'amélioration automatique de cas de test est un problème non linéaire, qui peut facilement se ramener à un problème d'optimisation de fonction (on veut trouver un ensemble de cas de test qui maximise le score de mutation pour un programme donné). Pour

cela, nous appliquons un algorithme semi-aléatoire particulièrement adapté à l'exploration de grands ensembles de solutions, l'algorithme génétique. Il existe une analogie forte entre le phénomène de sélection naturelle (duquel s'inspirent les algorithmes génétiques) et la génération de cas de test à partir d'un ensemble initial. Le problème d'optimisation de cet ensemble est modélisé avec de l'analogie suivante : un cas de test est considéré comme un prédateur et un mutant comme une proie. A partir de l'ensemble de test fourni par le testeur, le but de la sélection est alors de détecter les meilleurs prédateurs (cas de test), *i.e.*, ceux capables de tuer le plus de mutants, et de les améliorer d'une génération à l'autre. Nous présentons une modélisation de ce problème de test pour appliquer un algorithme génétique, ainsi que les résultats sur deux études de cas : une pour la génération de cas de test unitaire pour des classes Eiffel, et l'autre pour des cas de test système (un parseur pour le langage C# développé sur la plate-forme Microsoft.NET [MSDN"02a; MSDN"02b]).

Alors que les résultats de ces expériences s'avérèrent moins bons que ce que nous attendions, des collègues biologistes, du laboratoire ECOBIO de l'université de Rennes 1, nous suggérèrent d'adapter notre modèle pour nous concentrer sur l'adaptation au niveau bactériologique (des bactéries qui s'adaptent à un milieu déterminé) plutôt qu'au niveau animal (des lions qui tuent des zèbres). Une approche fondée sur une analogie avec le phénomène d'adaptation bactériologique [Rosenzweig"95] semble plus appropriée pour l'optimisation automatique de cas de test, en modifiant les algorithmes génétiques par l'ajout d'une fonction de mémorisation des meilleurs cas de test, et la suppression de la notion d'individu (animal) et de l'opération de croisement. Nous décrivons ce nouveau modèle, le *modèle bactériologique* et son comportement sur les études de cas précédentes. Nous détaillons aussi le paramétrage du modèle puisque c'est un des aspects essentiels pour l'efficacité de l'approche. Enfin, nous présentons une approche intermédiaire entre un algorithme génétique « pur » et un algorithme bactériologique « pur », issue des différentes expériences exécutées pour le calibrage.

Avant de présenter nos travaux sur l'étude d'algorithmes évolutionnistes pour l'optimisation automatique de cas de test, nous détaillons l'analyse de mutation. Cette technique, qui a été brièvement abordée dans le chapitre précédent, a en effet été fortement adaptée, et utilisée de façon intensive pour évaluer la qualité d'un ensemble de cas de test pour un composant.

3.1 Adaptation de l'analyse de mutation pour la qualification de composants

L'analyse de mutation est une technique proposée par DeMillo [DeMillo"78] pour valider et améliorer un ensemble de données de test. Le processus consiste à créer un ensemble de versions erronées du programme à tester appelées des *mutants*, puis à trouver un ensemble de données de test capable de détecter ces erreurs. Un vocabulaire particulier est associé à cette approche : si un cas de test peut détecter un mutant, on dit qu'il *tue le mutant*, sinon le *mutant*

est vivant. Un mutant est une copie exacte du programme sous test dans lequel une seule erreur simple a été injectée. En pratique, l'ensemble des mutants est créé en soumettant le programme à des *opérateurs de mutation* qui correspondent à différents types d'erreur (changement de signe des constantes, changement de sens d'une inégalité…).

Après une présentation du processus global pour l'analyse de mutation, nous détaillons plusieurs aspects de cette technique que nous avons adaptée pour fiabiliser des composants logiciels. En particulier, nous proposons une solution originale pour l'oracle, nous présentons des outils pour l'analyse de mutation développés au sein de l'équipe. Enfin, nous présentons l'architecture de notre approche qui permet d'exécuter une analyse de mutation pour une classe unitaire ou pour un système composé de plusieurs classe.

3.1.1 Le processus global

```
public IMoney addMoney(Money m) {
if (m.currency().equals(currency()) )
   return new Money(amount()+m.amount(), currency());
return new MoneyBag(this, m);}
```

Mutant 1

```
public IMoney addMoney(Money m) {
if (m.currency().equals(currency()) )
   return new Money(amount()+m.amount(), currency());
return new MoneyBag(this, m+1);}
```

Mutant 2

```
public IMoney addMoney(Money m) {
if (m.currency().equals(currency()) )

return new MoneyBag(this, m);}
```

Figure 9 – Exemples de mutant

Une analyse de mutation prend en entrée un programme sous test, et un ensemble de cas de test pour ce programme. Les mutants pour le programme sous test peuvent ensuite être générés automatiquement à partir de la définition d'un ensemble d'opérateurs de mutation. La Figure 9 illustre deux mutants qui peuvent être générés pour une méthode `addMoney()`. Le mutant 1 a été généré en ajoutant 1 à la variable m, le mutant 2 a été généré en supprimant une instruction.

Tous les cas de test associés au programme sont ensuite exécutés avec chacun des mutants. On obtient alors la *signature* de chaque cas de test (l'ensemble des mutants tués par un cas de test), et à partir de cette signature, le *score de mutation* de chaque cas de test (la proportion de mutants tués par un cas de test). Le processus est illustré Figure 10.

Signature d'un cas de test. *La signature d'un cas de test,* Sign(t)*, est l'ensemble des mutants non équivalents tués par ce cas de test:*

Sign(t) = {mutants tués par t}

Le score de mutation d'un cas de test est calculé à partir de sa signature.

Score de mutation. *Soit m le nombre total de mutants non équivalents d'un programme, le score de mutation de t est donné par la formule suivante:*

$$SM(t)=\frac{card(Sign(t))}{m}$$

On peut aussi calculer la signature et le score de mutation d'un ensemble de tests.

Valeurs globales d'un ensemble de tests. *La signature globale d'un ensemble $T = \{t_1 \ldots t_n\}$ de cas de test est l'union des signatures de tous les cas de test de l'ensemble:*

$$sign(T)=\bigcup_{i=1}^{card(T)} Sign(t_i)$$

et le score global est: $SM(T)=\dfrac{card(Sign(T))}{m}$

Le score de mutation associé à un ensemble de cas de test correspond à l'estimateur de la qualité de cet ensemble de cas de test.

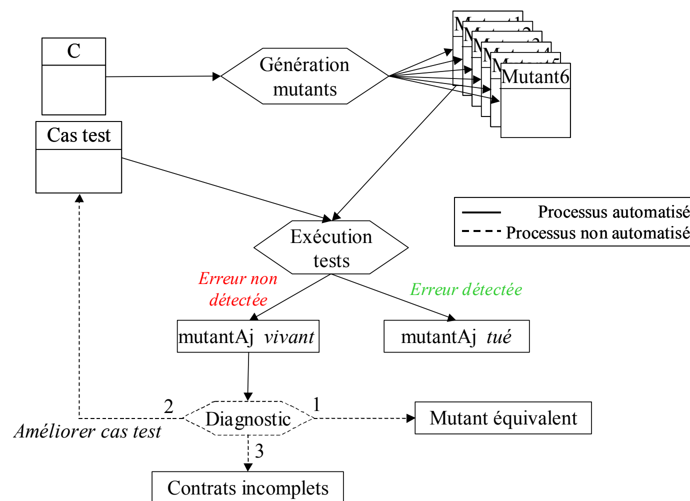


Figure 10 – Processus global de génération de tests par mutation

3.1.2 L'oracle pour l'analyse de mutation

Lors de l'exécution des cas de test sur l'ensemble des mutants, il faut une fonction d'oracle qui évalue si le cas de test tue le mutant. Lorsque nous exécutons des analyses de mutation dans la suite de cette thèse, nous utilisons deux types d'oracle. La première fonction d'oracle consiste à repérer une différence de comportement entre le mutant et le programme original. Elle correspond à la fonction la plus généralement utilisée pour la mutation. L'idée consiste à dire que, puisque les cas de test sont censés avoir détecté les erreurs du programme, et puisqu'il n'y a aucun moyen d'assurer la qualité du programme initial, on considère ce dernier comme correct et on teste les cas de test en les mettant à l'épreuve sur des versions volontairement erronées du programme original. La seconde fonction utilise les assertions

embarquées dans le code, si elles existent. Dans les cas des expériences décrites par la suite (chapitre 4), nous utilisons les contrats comme valeur d'oracle. Ceci constitue une contribution originale de cette thèse.

Oracle par différence de comportement. *Soit* Out_o *l'ensemble des sorties obtenues en exécutant un cas de test sur le programme original,* Out_m *l'ensemble des sorties obtenues en exécutant un cas de test sur un mutant. Si* $Out_o \neq Out_m$*, alors le mutant est tué par le cas de test. C'est ce qu'on appelle la fonction d'oracle par différence de comportement.*

Oracle fondé sur les assertions. *Si au cours de l'exécution d'un cas de test sur un mutant, une assertion est violée, on considère que le cas de test a détecté le mutant.*

La fonction d'oracle par différence de comportement est la fonction généralement utilisée pour l'analyse de mutation. Dans ce cas, la génération de test se fait de manière itérative. Le testeur exécute une première analyse de mutation en faisant l'hypothèse que le programme original est correct. Il obtient alors un ensemble de cas de test, puis il exécute ces cas de test sur le programme original et des erreurs peuvent alors être détectées. Lorsqu'elles sont corrigées, un seconde analyse de mutation est exécutée puisque le programme original a changé, et donc la fonction d'oracle également. De nouveaux cas de test sont ainsi obtenus.

Dans cette thèse, nous utilisons deux notions de « sortie » pour comparer les comportements. La première, « classique », est fondée sur la trace d'exécution du programme. La seconde, spécifique à l'objet, est fondée sur la différence entre les objets du programme initial et les objets du programme mutant. Dans ce cas, la différence de comportement est vérifiée en comparant les valeurs des attributs des objets, après l'exécution d'un cas de test. Cette seconde manière de fabriquer l'oracle est une contribution technique secondaire de ce manuscrit. Cela permet de s'affranchir des traces explicites, et permet notamment d'avoir un oracle efficace, même dans le cas de programmes ayant peu de sorties explicites. On peut noter que l'observation de la différence de comportement est considérée comme un « oracle » par abus de langage, puisqu'en pratique on ne sait pas si le cas de test a détecté une faute non intentionnelle dans le programme initial.

Nous verrons par la suite comment utiliser l'oracle fondé sur les assertions. Il est employé dans deux contextes : l'amélioration des contrats embarqués dans le composant (sectoin 4.2.5a), et l'évaluation de l'apport des assertions pour la robustesse et la diagnosabilité d'un composant logiciel (section 4.2.5b).

Enfin, nous utilisons un troisième type d'oracle, en associant des assertions aux cas de test (comme le requiert le format JUnit par exemple). C'est la cas le plus classique pour l'expression de l'oracle.

A la fin d'une analyse de mutation, un diagnostic doit être fait pour savoir pourquoi certains mutants n'ont pas été tués. Trois raisons peuvent empêcher la détection d'un mutant :

aucune donnée de test ne permet d'exhiber une défaillance lors de l'exécution de l'instruction erronée, les assertions dans le code sont trop « faibles » pour détecter l'erreur, ou le mutant est équivalent au programme original. Dans le premier cas, soit aucun cas de test ne permet de couvrir l'instruction erronée, soit la donnée de test n'a pas permis de produire l'erreur en exécutant l'instruction mutée. Dans le cas où on utilise un oracle fondé sur les assertions, un mutant peut ne pas être tué à cause d'une incomplétude des assertions. La section suivante détaille le problème des mutants équivalents.

Remarque : les deux premières raisons ne sont pas exclusives l'une de l'autre : même si un cas de test exécute l'instruction erronée et détecte ainsi l'erreur, les assertions peuvent être trop faibles pour la détecter.

3.1.3 Mutants équivalents

Parmi les mutants générés, certains sont *équivalents* au programme initial.

Mutant équivalent. *Soit D_i le domaine des valeurs d'entrée du programme à tester, on dit qu'un mutant est équivalent au programme initial s'il n'existe aucune séquence de valeurs de D_i qui permette de distinguer le programme mutant du programme initial. Soit $Out_o(x_1,\ldots x_n)$ les sorties du programme original pour l'ensemble des entrées $(x_1,\ldots x_n)$ et $Out_m(x_1,\ldots x_n)$ les sorties pour un mutant, le mutant est dit équivalent si :* $\neg(\exists(x_1,\ldots x_n)\in D_i)\ /\ Out_m(x_1,\ldots x_n) \neq Out_o(x_1,\ldots x_n)$

Sur l'exemple de la Figure 11, la variable I a été remplacée par MinVal dans la seconde instruction du programme mutant. Or, l'instruction précédente affecte la valeur I à MinVal. A ce point du programme MinVal et I ont donc toujours la même valeur. Le programme mutant est équivalent au programme initial.

Le programme original

```
Function Min (I , J : integer)
          return integer IS
   MinVal : integer;
Begin
  Minval:=I;
  if (J < I) then MinVal:=J;
  return (MinVal)
End;
```

Le programme mutant

```
Function Min (I , J : integer)
          return integer IS
   MinVal : integer;
Begin
  Minval:=I;
  if (J < MinVal) then MinVal:=J;
  return (MinVal)
End;
```

Figure 11 – Exemple de mutant équivalent

Il est important de détecter les mutants équivalents et de les éliminer de l'ensemble de mutants à tuer. En effet, par définition, il est impossible d'écrire un cas de test qui distingue un mutant équivalent du programme initial, et les mutants équivalents ne peuvent donc pas être tués. Ils sont en fait corrects. Dans [Offutt"97], les auteurs proposent une technique pour détecter automatiquement certains types de mutants équivalents. L'idée consiste à ramener le problème de la génération de test à un problème de résolution de contrainte. Pour tuer un

mutant, il faut générer une donnée de test qui satisfasse une contrainte particulière. Si la contrainte ne peut être satisfaite alors le mutant est équivalent.

Cependant, cette détection se fait généralement à la main, ce qui augmente le coût de l'analyse de mutation. De manière pratique, on écrit un premier ensemble de cas de test qui tue un sous-ensemble des mutants. On cherche ensuite les mutants équivalents parmi ceux encore vivants, ce qui évite d'observer tous les mutants générés.

3.1.4 Opérateurs de mutation utilisés pour la qualification des composants

Dans notre étude, nous appliquons l'analyse de mutation à la qualification de tests pour des programmes objet. Certains opérateurs sont donc liés à ce type de programme, d'autres sont des opérateurs couramment utilisés par les différents outils de mutation. Tous les opérateurs sont donnés dans la Tableau 1. Au moment de notre étude, les opérateurs proposés étaient originaux. Entre-temps, Offutt [Ma"02] et Kim [Kim"01] ont proposé de nouveaux opérateurs pour les programmes OO, qu'il nous a, malheureusement, été impossible d'utiliser.

Type	Description
EHF	Exception Handling Fault
AOR	Arithmetic Operator Replacement
LOR	Logical Operator Replacement
ROR	Relational Operator Replacement
NOR	No Operation Replacement
VCP	Variable and Constant Perturbation
MCR	Methods Call Replacement
RFI	Referencing Fault Insertion

Tableau 1 – Opérateurs de mutation pour les programmes objet

EHF : Insertion d'une instruction d'exception qui est systématiquement déclenchée lorsqu'elle est exécutée.

AOR : Remplace les occurrences de "+" par "-" et vice et versa.

Opérateur arithmétique	Remplacé par
+	-, *
-	+, / (ou div)
*	/ (ou div), +
/	*, -
div	-, mod
mod	-, div

LOR : Chaque occurrence d'un opérateur logique (et, ou, non-et, non-ou, ou exclusif) est remplacée par chacun des autres opérateurs. De plus, l'expression est remplacée par VRAI et FAUX.

Opérateur relationnel	Remplacé par
<	≤
>	≥
≠	< et >
≤	<
≥	>
=	≤ et ≥

ROR : Chaque occurrence d'un opérateur relationnel (<, >, ≤, ≥, =, ≠) est remplacée par chacun des autres opérateurs. Pour éviter d'avoir un trop grand nombre de mutants, on applique les règles suivantes présentées dans le tableau ci-dessus.

NOR : Supprime une instruction.

VCP : Les valeurs des constantes et des variables sont légèrement modifiées pour effectuer une analyse de sensibilité (*sensitivity analysis*) analogue à ce que propose Voas dans [Voas"92a]. Chaque constante ou variable de type arithmétique est incrémentée ou décrémentée de un. Chaque booléen est remplacé par son complément.

MCP : Les appels de méthode sont remplacés par un appel à une autre méthode qui a la même signature.

RFI : Force à *void* la référence à un objet après sa création. Supprime une instruction de clonage ou de copie. Introduit une instruction de clonage pour chaque affectation d'une référence.

Les opérateurs de mutation AOR, LOR, ROR et NOR sont des opérateurs de mutation traditionnels étudiés dans [DeMillo"91; Offutt"96a]. Nous avons introduits les autres opérateurs pour le test dans le domaine des programmes objet. L'opérateur RFI injecte des fautes de référence à des objets (fautes d'aliasing) qui sont spécifiques au domaine de la programmation objet :

- la référence à un objet est forcée à *void* (en Eiffel) ou *null* (en Java et C#).
- les instructions de duplication d'objets sont supprimées
- chaque affectation d'une référence à un objet est précédée de la duplication de cet objet

Les fautes introduites par l'opérateur RFI sont plus difficiles à détecter que celles introduites par les autres opérateurs.

3.1.5 Outils

Trois outils pour l'analyse de mutation ont été développés au sein de l'équipe Triskell, chacun d'entre eux étant destiné au test de programmes dans un langage particulier. Les trois langages étudiés sont des langages orientés objet : Eiffel, Java et C#. Les principes de

fonctionnement de ces outils sont similaires : ils prennent tous les trois un programme sous test et un ensemble de cas de test en entrée. Ensuite l'outil génère les mutants automatiquement. Un mode interactif permet à l'utilisateur de paramétrer la génération : uniquement les mutants pour un ou plusieurs opérateurs particuliers ou un nombre déterminé de mutants au hasard parmi tous les mutants possibles ou encore, tous les mutants possibles. Une fois que l'ensemble de mutants sur lequel on veut lancer l'analyse de mutation est déterminé, l'outil exécute chaque cas de test sur chaque mutant et calcule la signature des cas de test ainsi que le score de mutation global.

L'outil pour Eiffel est baptisé µSlayer [Deveaux"99b] et permet d'exécuter une analyse de mutation pour une classe Eiffel. Il implante tous les opérateurs décrits dans la section 3.1.4. Les cas de test en entrée de µSlayer doivent être regroupés dans une seule classe. Lors de l'analyse de mutation, toutes les méthodes de cette classe de test sont exécutées avec chaque mutant.

L'outil pour Java est baptisé JMutator [Deveaux"01] et permet d'exécuter une analyse de mutation sur une ou plusieurs classes Java. Il implante tous les opérateurs décrits dans la section 3.1.4. Cet outil s'appuie sur deux outils pour Java : JavaCC (Java's Compiler Compiler) [SUN"00] et JUnit [Beck"01]. Le parseur de JMutator, nécessaire à la génération des mutants, a été généré automatiquement avec JavaCC. Les cas de test en entrée de JMutator doivent être écrits au format JUnit, ce qui permet d'utiliser les mécanismes propres au framework JUnit pour exécuter les cas de test et récupérer la cause de l'erreur.

Enfin, l'outil pour les programmes C#, baptisé NMutator, a pour particularité d'être dédié à l'analyse de mutation sur des systèmes orientés objet. Comme nous le verrons au chapitre 3, ceci implique de ne générer que certains types de mutants, NMutator n'implante donc que les opérateurs NOR et AOR. Le parseur pour cet outil est inspiré du parseur de Mono, un projet de compilateur pour C# pour linux [de Icaza"01]. Par ailleurs, NMutator permet la parallélisation de l'exécution des cas de test sur différents processus.

Les outils JMutator et NMutator sont disponibles en ligne à l'adresse suivante : http://franck.fleurey.free/index.htm.

3.1.6 Analyse de mutation pour une classe ou un système

Dans la suite de ce chapitre, nous proposons d'optimiser automatiquement un ensemble de cas de test pour une classe ou un système composé d'un ensemble de classes, et nous utilisons l'analyse de mutation pour qualifier ces cas de test. Cette technique est généralement appliquée au niveau unitaire, et un mutant est alors une classe avec une erreur. Dans le cas du test système, un mutant est une copie du système sous test dans laquelle on a injecté une erreur.

Au moment du test système, on suppose que la phase de test unitaire a permis de valider les classes séparément, et on se concentre sur le test des interactions entre classes. Les deux étapes de test ne cherchant pas à détecter le même type d'erreur, l'analyse de mutation doit

être adaptée aux deux situations. De plus, des considérations techniques doivent être prises en compte pour appliquer la mutation à un système. Alors qu'il est raisonnable d'injecter de nombreuses erreurs dans une classe, ce n'est pas raisonnable au niveau système. En effet, les temps de compilation et d'exécution pour appliquer tous les cas de test sur un système mutant sont beaucoup plus important que pour une classe isolée. De plus, s'il est possible de détecter les mutants équivalents pour une classe dans la plupart des cas (le plus souvent à la main), il est beaucoup plus difficile de détecter un système équivalent.

Pour appliquer l'analyse de mutation à la génération de cas de test système, nous avons donc choisi un sous-ensemble d'opérateurs pour générer moins de mutants. De plus, ces opérateurs ne doivent pas générer de mutants équivalents. Pour les études de cas présentées dans ce chapitre, nous avons choisi les opérateurs LOR (logical operator replacement) et NOR (No operation replacement) pour le test système. La Figure 12 présente l'architecture générique pour un générateur de mutant : dans le cas du test unitaire, le générateur utilise tous les opérateurs définis précédemment (qui implantent l'interface OPERATEURUNITAIRE), alors que dans le cas du tes système, seuls les opérateurs LOR et NOR (qui implantent l'interface OPERATEURSYSTEME) sont utilisés.

Une autre solution pour éviter de générer trop de mutants, que nous n'avons pas eu le temps d'explorer au cours de cette thèse, consiste à sélectionner les classes dans lesquelles des erreurs doivent être injectées en fonction d'un certain critère. Plusieurs critères semblent possibles : uniquement les classes de contrôle, les classes les plus éloignées de l'entrée du système…

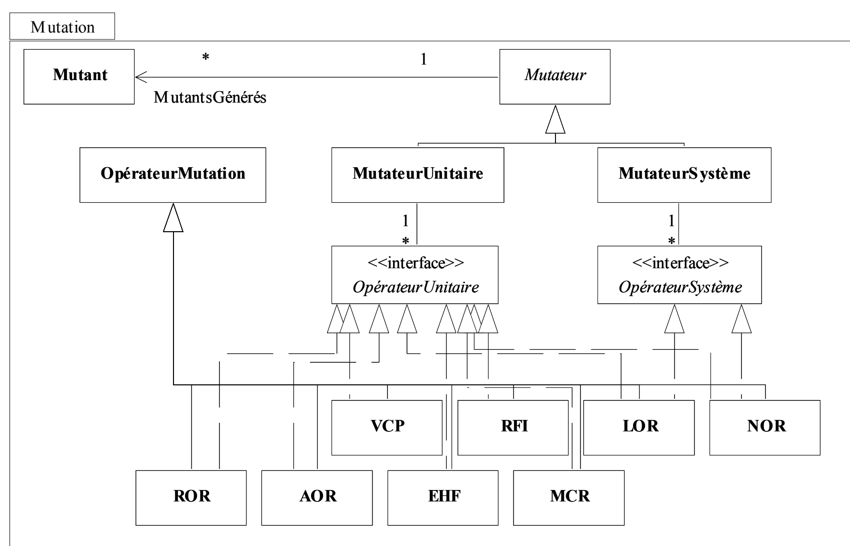


Figure 12 – Architecture pour la génération de mutants

3.1.7 Optimisation automatique et analyse de mutation

Pour conclure cette section sur l'analyse de mutation, et introduire la problématique principale de ce chapitre, la Figure 13 illustre le processus de mutation dans lequel la phase d'amélioration de cas de test est automatisée. Dans le cas où l'ensemble de cas de test n'est pas assez efficace, l'opérateur Optimiseur essaie de l'améliorer automatiquement. Dans la suite de ce chapitre nous étudions deux algorithmes évolutionnistes pour implanter cet opérateur.

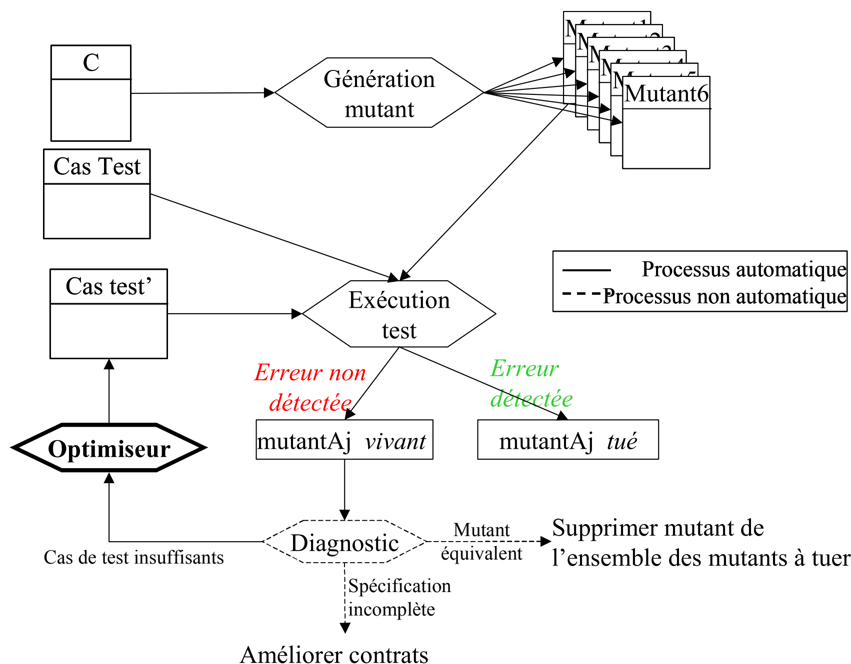


Figure 13 – Amélioration automatique de cas de test pour l'analyse de mutation

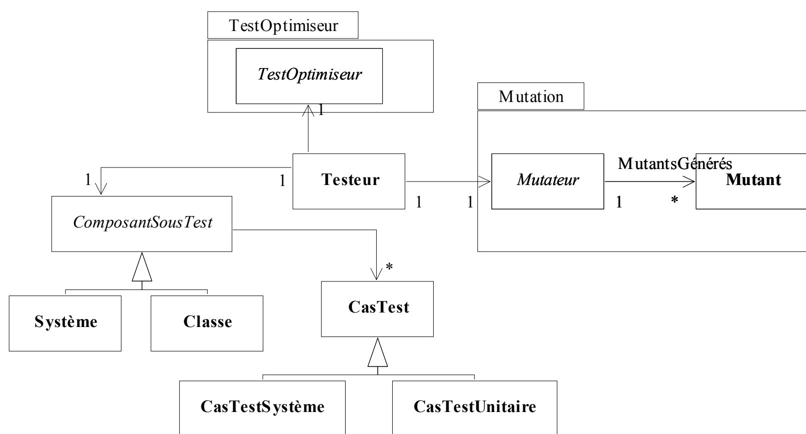


Figure 14 – Architecture globale pour la génération de test

La Figure 14 illustre l'architecture UML globale pour la génération automatique de test fondée sur l'analyse de mutation pour la qualification de cas de test. La classe TESTEUR est chargée de la connection entre le COMPOSANTSOUSTEST, le générateur de mutants représenté par la classe MUTATEUR, et le TESTOPTIMISEUR. Dans ce chapitre, nous nous intéressons à l'optimisation de cas de test pour une CLASSE ou un SYSTEME. A un composant sous test, on associe un ensemble de cas de test. Dans les sections 3.2.3b et 3.2.3c, nous détaillons la modélisation d'un CASTESTSYSTEME ou CASTESTUNITAIRE.

3.2 Algorithme génétique pour l'optimisation de cas de test

La confiance que l'on peut avoir en un composant dépend de la fiabilité de ses tests et de ses contrats. Une étude récente [Deveaux"99a] a montré qu'il est facile, pour un testeur, d'écrire à la main des tests obtenant un score de mutation entre 50 et 70%, mais l'obtention d'un score plus élevé réclame un effort important. Le but de notre étude est donc d'automatiser l'optimisation d'un ensemble initial de tests.

Dans cette section, nous proposons une première méthode pour résoudre ce problème. Cette approche est fondée sur un algorithme génétique qui devra optimiser le score de mutation de l'ensemble initial de tests. Nous commençons par présenter globalement les algorithmes génétiques, puis la manière dont nous les avons appliqués à notre problème. Nous présentons ensuite les expériences que nous avons effectuées avec ce modèle, et les conclusions que nous en avons tirées.

3.2.1 Les algorithmes génétiques

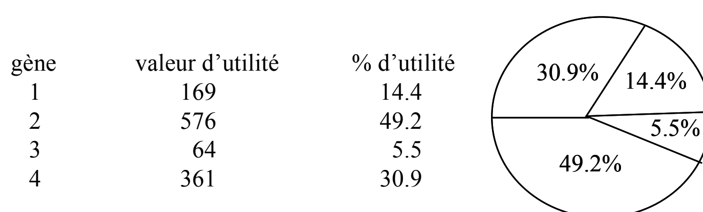
Les algorithmes génétiques [Goldberg"89] ont d'abord été développés par John Holland [Holland"74] dont le but était d'expliquer les systèmes naturels et de concevoir des systèmes artificiels fondés sur ces mécanismes naturels. Les algorithmes génétiques sont donc des algorithmes d'optimisation fondés sur les principes de la sélection naturelle. Dans la nature, les individus les mieux adaptés à leur environnement (qui sont capables d'échapper aux prédateurs, de se protéger du froid…) se reproduisent, et grâce aux croisements et à la mutation, la génération suivante sera, normalement, encore mieux adaptée. C'est exactement ainsi que fonctionnent les algorithmes génétiques : à partir d'un critère objectif ils permettent de sélectionner les meilleurs individus qui se reproduiront pour fournir la génération suivante.

Les algorithmes génétiques sont relativement simples à programmer, et sont particulièrement utiles lorsque l'espace de recherche d'une solution est très grand et qu'il existe des optima locaux. De plus, la recherche d'un optimum se fait parmi un ensemble de solutions possibles et non pas en observant le comportement de solutions isolées. En effet, un algorithme génétique part d'un ensemble initial de solutions, souvent créées aléatoirement, dans lequel il sélectionne celles qui sont les plus proches de l'optimale. A partir de cet échantillon, il crée de nouvelles solutions par croisement et mutation. Comme les nouvelles solutions sont créées à partir des meilleures de la génération précédente, elles ont des chances d'être meilleures que leurs ancêtres.

Pour appliquer un algorithme génétique à un problème particulier, il faut décomposer celui-ci en unités atomiques qui correspondent à des *gènes* pour le modèle. Des *individus* sont ensuite construits en une suite ordonnée de gènes. Les individus pour une modélisation particulière, ont tous la même taille (le même nombre de gènes). Un ensemble d'individus pour l'algorithme est appelé une *population*. Pour guider la sélection, il faut définir une *fonction d'utilité* U qui, pour chaque individu d'une population, rend une valeur U(x) qui correspond à la qualité de l'individu par rapport au problème que l'on veut résoudre. Cette fonction est le critère à maximiser sur la population de départ. De plus, un algorithme génétique utilise trois opérations:

- la *reproduction* a pour but de sélectionner les individus d'une population qui vont participer à la génération suivante. Cette opération consiste en un tirage au sort pour lequel les individus sont pondérés par leur valeur d'utilité. Ce tirage revient au lancement d'une roulette où chaque individu aurait une part proportionnelle à sa valeur d'utilité.

  Sur l'exemple de la Figure 15, on voit bien qu'en lançant cette roulette, les individus possédant la plus grande valeur d'utilité ont plus de chances d'être tirés, mais la probabilité de tirer un individu moins bon n'est pas nulle: ils peuvent contenir quelques gènes bénéfiques à la population.



gène	valeur d'utilité	% d'utilité
1	169	14.4
2	576	49.2
3	64	5.5
4	361	30.9

Figure 15 – Fonctionnement de l'opérateur de reproduction

- le *croisement* consiste à tirer au hasard un nombre k positif inférieur à la taille n des individus. Ensuite, à partir de deux individus A et B deux autres individus sont créés, l'un formé des k premiers gènes de A et de n-k derniers gènes de B, l'autre formé des n-k derniers gènes de A et des k premiers gènes de B. Il existe d'autres opérateurs de croisement, mais celui décrit ici est le plus couramment utilisé.
- la *mutation* consiste à modifier la valeur d'un ou plusieurs gènes d'un individu.

Exemple : si un individu est une suite de gènes codés chacun sur un bit, la mutation d'un gène consiste à lui donner la valeur 0 si sa valeur initiale était 1, ou 1 sinon.

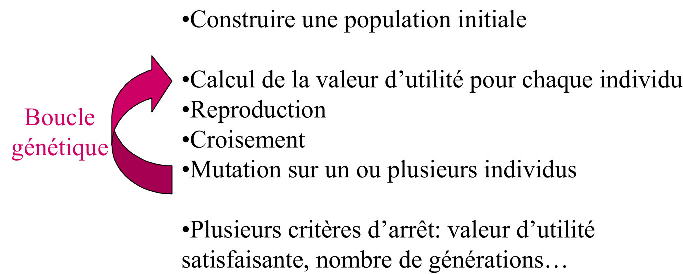


Figure 16 – Processus global pour un algorithme génétique

Habituellement, les opérateurs de reproduction et surtout de croisement sont si importants pour la convergence vers la meilleure solution (sélection parmi les meilleurs et croisement de ces individus), que l'opérateur de mutation joue un rôle secondaire (le taux de mutation à une génération tourne typiquement autour de 2%). Le processus global de l'algorithme est décrit Figure 16 et le texte est détaillé dans la section suivant.

3.2.2 Texte de l'algorithme

Pour la description du texte de l'algorithme génétique, on suppose l'existence des fonctions `select`, `rand(Entier, Entier)` et `rand(Population)`. L'implantation des fonctions `Utilité` et `Mute` dépend de l'application particulière de l'algorithme génétique. Toutes ces fonctions sont décrites brièvement avant l'algorithme génétique.

```
// La fonction select prend une population et une séquence de réels en entrée.
La séquence de réels doit être de la même taille que la population. La
fonction rend un individu tiré au hasard dans la population, un individu i
ayant une probabilité proba[i] d'être tiré.
select(Population pop, Ens[Réel] proba) : Individu

//La fonction rand rend un entier au hasard entre x et y
rand(Entier x, y) : Entier

//La fonction rand rend un individu au hasard dans pop
rand(Population pop) : Individu

//La fonction Utilité rend la valeur d'utilité d'un individu
Utilité (Individu ind) : Réel

//La fonction Mute rend l'individu ind dans lequel un gène a été muté
Mute (Individu ind) : Individu

//Un algorithme génétique prend une population initiale, un objectif et un
taux de mutation en entrée. Il explore l'espace des solutions jusqu'à trouver
un individu dont l'utilité est supérieure ou égale à l'objectif
Algo_génétique (Population P, Réel obj, Réel tx_mutation) : Individu
  local Population P_rep, P_crois, P_mut ;
  début
    tantque ( ! (∃ i ∈ [0..P.taille-1] | Utilité(P[i]) > obj)
    faire
      P_rep ← ∅ ; P_crois ← ∅ ; P_mut ← ∅ ;
      P_rep ← Reproduction(P) ;
      P_crois ← Croisement (P_rep) ;
      P_mut ← Mutation(P_crois, tx_mutation) ;
```

```
      P ← Pmut ;
    fait
    Result ← (P[i] | Utilité(P[i]) > obj ;
  fin

Reproduction (Population pop) : Population
  local Ens[Réel] proba ;
      Réel u_cumulée ;
      Population poptmp ;
  début
    u_cumulée ← Σ(i=0..pop.taille) Utilité(pop[i]) ;
    ∀i ∈ [O..pop.taille-1] proba[i] ← pop[i] * (Utilité(pop[i])/u_cumulée) ;
    poptmp ← ∅ ;
    pour i de 0 à pop.taille-1 faire
      poptmp ← poptmp ∪ select(pop, proba);
    fait
    Result ← poptmp ;
  fin

Croisement (Population pop) : Population
  local Ens[Individu] croisement ;
      Population poptmp ;
  début
    poptmp ← ∅ ;
    pour i de 0 à pop.taille-1 faire
      croisement ← Croise(pop [i], pop [i+1]) ;
      poptmp ← croisement[0] ∪ croisement[1] ∪ poptmp ;
    fait
    Result ← poptmp ;
  fin

Croise(Individu ind1, ind2) : Ens[Individu]
  local Entier n ;
        Individu ind3, ind4 ;
  début
    taille ← ind1.taille ;
    n ← rand(0, taille-1) ;
    ind3 ← ind1[O..n] ∪ ind2[n+1..taille-1];
    ind4 ← ind2[0..n] ∪ ind1[n+1..taille-1];
    Result ← ind3 ∪ ind4
  fin

Mutation (Population pop, Réel tx_mutation) : Population
  local Population poptmp ;
  début
    poptmp ← ∅ ;
    pour i de 0 à (pop.taille * tx_mutation) - 1 faire
      pop ← pop \ rand(pop) ;
      poptmp ← Mute(ind) ∪ poptmp ;
    fait
    Result ← pop ∪ poptmp ;
  fin
```

3.2.3 Le problème de l'optimisation de cas de test

Cette section détaille la modélisation du problème de l'optimisation automatique de cas de test pour appliquer un algorithme génétique. Nous présentons d'abord un modèle générique pour l'optimisation de n'importe quel type de donnée de test. Nous discutons ensuite des problèmes et adaptations du modèle spécifiques à la génération des cas de test unitaires ou système.

a Modélisation générique des algorithmes génétiques pour le test

La Figure 17 présente une architecture générique pour l'optimisation automatique de cas de test à l'aide d'un algorithme génétique. Cette architecture est générique en ce sens qu'elle peut être spécialisée pour une phase de test donnée. Ici, nous proposons deux axes de spécialisation que sont le test unitaire de classe et le test système (ou d'un paquetage de classes). Un algorithme génétique est une technique pour optimiser un ensemble de cas de test, la classe GENETIQUE hérite donc de TESTOPTIMISEUR. Le problème est décomposé en une population qui est composée d'individus eux-mêmes constitués d'un ensemble ordonné de gènes (cf. section 3.2.1). Les tailles de la population et des individus sont des constantes fixées pour une application particulière d'un algorithme génétique.

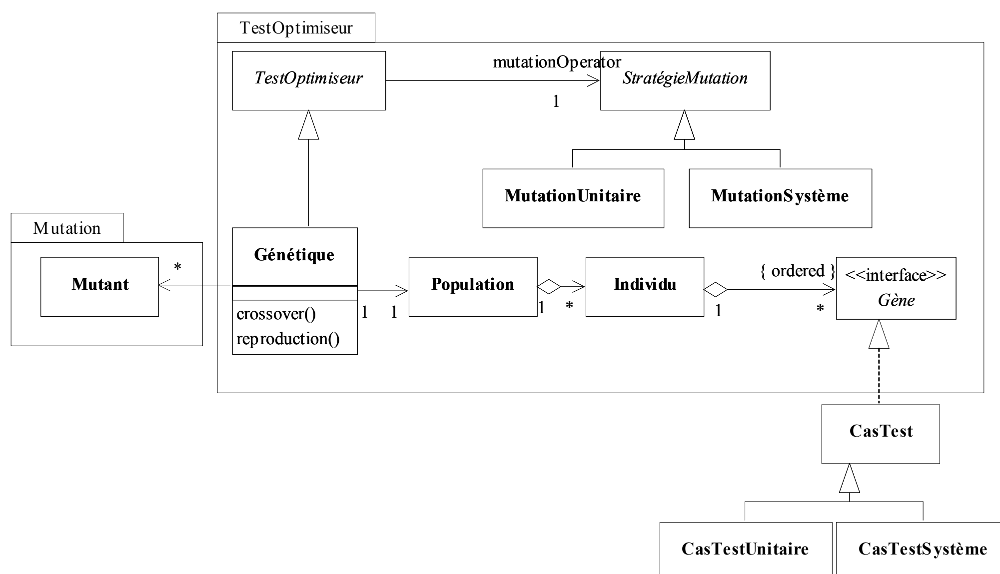


Figure 17 – Une architecture générique pour l'optimisation de cas de test à l'aide d'un algorithme génétique

La modélisation d'un *gène* est particulière au problème de l'optimisation de cas de test : il correspond à un cas de test (CASTEST implémente l'interface GENE). Il apparaît sur le diagramme que la classe CASTEST se spécialise en CASTESTUNITAIRE et CASTESTSYSTEME. Nous détaillons les particularités de modélisation pour les cas de test unitaires ou systèmes dans les sections suivantes.

Cette modélisation du gène implique qu'un individu est un ensemble de cas de test. Un individu correspond donc bien à ce que l'on veut optimiser ici : un ensemble de cas de test pour un système ou pour une unité (une classe dans un cadre orienté objet).

Un autre paramètre doit être modélisé pour une application particulière de l'algorithme génétique : la *fonction d'utilité*. Nous avons choisi le score de mutation comme mesure de l'utilité d'un individu. La classe GENETIQUE encapsule donc l'ensemble de mutants sur lequel le score de mutation sera calculé.

Enfin, il faut définir les opérateurs pour l'algorithme. Les opérations de reproduction et de croisement peuvent être définies de manière générique, puisque la modélisation des individus ne dépend pas d'un type particulier de cas de test. Par contre, comme l'opération de mutation s'effectue sur les gènes, et que la forme de ceux-ci dépend du type de cas de test généré, cette opération sera détaillée dans les sections spécifiques à chaque type de cas de test. Cette distinction entre les opérations apparaît sur le diagramme de la Figure 17 : les opérations de reproduction et de croisement apparaissent directement dans la classe GENETIQUE, alors que l'opération de mutation est réifiée en MUTATIONUNITAIRE et MUTATIONSYSTEME.

- Reproduction : les individus sont reproduits en prenant leur score de mutation comme valeur d'utilité.
- Croisement : soit m la taille des individus d'une population, et i un entier tiré au hasard entre 1 et m-1. Alors, à partir de deux individus ind_1 et ind_2, deux nouveaux individus sont créés, le premier avec les i premiers gènes de ind_1 et les m-i derniers gènes de ind_2, et l'autre avec les i premier gènes de ind_2 et les m-i derniers de ind_1. Cet opération est illustrée sur la Figure 18.

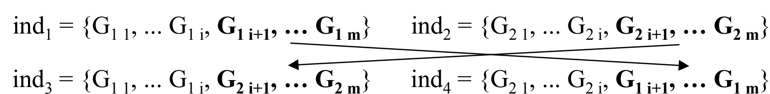


Figure 18 – Opérateur de croisement

Les sections suivantes détaillent les aspects de la modélisation dépendant du type de cas de test à générer : le modèle de gène et l'opérateur de mutation. Dans les deux cas, un gène doit être un cas de test pour rester cohérent avec le modèle de la Figure 17. Par ailleurs, le modèle pour un gène doit être correct pour les autres opérations, notamment le croisement. En effet, au cours de cette opération, il faut pouvoir changer l'ordre des gènes à l'intérieur d'un individu, et que celui-ci reste une entrée valide pour le programme sous test.

b Spécialisation pour le test unitaire

Nous considérons que la classe est l'unité pour le test unitaire de système orienté objet. Un cas de test pour une classe unitaire est alors une méthode qui crée une ou plusieurs instances de la classe sous test, et appelle des méthodes sur ces objets. Les cas de test peuvent être ensuite regroupés dans une classe de test. Cette structuration des cas de test unitaire est

largement adoptée depuis l'émergence des frameworks de test XUnit disponibles pour de nombreux langages (http://www.junit.org/index.htm).

Pour appliquer un algorithme génétique pour l'optimisation automatique de cas de test unitaires, nous avons modélisé le gène par une méthode qui correspond à un cas de test unitaire dans une classe de test. Deux parties peuvent être clairement identifiées dans le corps de cette méthode : l'initialisation qui consiste à créer des instances de la classe sous test (correspond au setup dans les frameworks XUnit), et l'appel de méthodes sur ces objets. La Figure 19 illustre un exemple de gène pour la génération automatique de cas de test unitaire pour une classe Eiffel. La méthode `test_set` de la classe TEST_DATE est un gène, et les parties initialisation et appel de méthodes sont séparées par la balise `--MethodCalls`.

```
class TEST_DATE

feature

  test_set is
    local date:P_DATE;
  do
      !!date;
      --MethodCalls
      date.set_year(1999);
      date.set_month(7);
      date.set_day(2);
    end

end
```

Figure 19 – Exemple de gène pour la génération de cas de test unitaire

Modèle de gène pour le test unitaire. *Un gène est un cas de test unitaire. Il est modélisé par une méthode composée de deux parties. Soit* $m_1,\ldots,m_n$ *n méthodes appelées et* $p_1,\ldots,p_n$ *n ensembles de paramètres effectifs pour les appels de méthode. Un gène est défini par une paire I, S (I pour initialisation, et S pour la séquence d'appels) telle que* $G=[I, S]$ *où* $S=(m_1(p_1),\ldots,m_n(p_n))$.

1. *I : création et initialisation d'instances de la classe sous test, et d'objets nécessaires au test*
2. *S : séquence d'appels de méthodes sur les objets créés*

L'opération de mutation pour l'algorithme génétique peut être définie à partir de ce modèle de gène. Elle consiste à modifier les valeurs des paramètres effectifs pour un appel de méthode dans le gène.

Opération de mutation pour le test unitaire. *L'opération de mutation de l'algorithme génétique modifie les paramètres effectifs d'un appel de méthode d'un gène, selon la règle suivante :* $G = [I , S] \Rightarrow G = [I , S_{mut}]$

$$S=(m_1(p_1),\ldots, m_i(p_i),\ldots, m_n(p_n)) \Rightarrow S_{mut}=(m_1(p_1),\ldots, m_i(p_{imut}),\ldots, m_n(p_n))$$

Cette opération est importante pour générer des données qui permettent de couvrir d'autres parties du code. Par exemple dans le cas d'une conditionnelle dans une méthode, il faut appeler cette méthode avec au moins deux valeurs différentes, pour pouvoir passer dans les deux branches de la conditionnelle. La Figure 20 illustre un exemple pour l'opération de mutation sur un gène. Dans ce cas, la valeur `1999` du premier paramètre pour l'appel de la méthode `set` est remplacée par `1998`.

Les définitions du gène et de l'opération de mutation données ici, correspondent aux concepts de CASTESTUNITAIRE et MUTATIONUNITAIRE du diagramme de la Figure 17.

<pre>class
                UNIT_TEST_EXAMPLE
inherit
                EUNIT_TESTCASE
feature -- Support
  date:P_DATE;
 	
  set_up is
    do
      !!date.make (10)
    end
 	
feature -- Tests
  test_comparison is
    local date1:P_DATE;
    do
      !!date1;
      date.set(**1999**,7,5);
      date1.set(1998,7,5);
    end
 	
end – UNIT_TEST_EXAMPLE</pre>
	<pre>class
	                UNIT_TEST_EXAMPLE
	inherit
	                EUNIT_TESTCASE
	feature -- Support
	  date:P_DATE;
	
	  set_up is
	    do
	      !!date.make (10)
	    end
	
	feature -- Tests
	  test_comparison is
	    local date1:P_DATE;
	    do
	      !!date1;
	      date.set(**1998**,7,5);
	      date1.set(1998,7,5);
	end
	
	end – UNIT_TEST_EXAMPLE</pre>

Figure 20 – Exemple de mutation d'un gène pour la génération de cas de test unitaires

c Spécialisation pour le test système

Cette section détaille le modèle de gène et l'opérateur de mutation associé dans le cas de l'optimisation de cas de test systèmes. La modélisation d'un cas de test système est fortement liée au format d'entrée du système. Par exemple, pour l'étude de cas sur un parseur C#, un cas de test est un fichier source C#, qui doit être syntaxiquement correct pour la grammaire du langage. La difficulté pour appliquer l'algorithme génétique dans ce cas, était donc de pouvoir générer automatiquement des programmes correct, et pour les expériences, il a fallu écrire un générateur spécifique au langage C#. Cependant, ce générateur de données de test pour un système pourrait être ré-utilisé pour d'autres études de cas, si les données en entrée peuvent être spécifiées à l'aide d'une grammaire.

Modèle de gène pour le test système. *Dans le cas particulier d'un parseur, un gène est un fichier source écrit dans le langage particulier. Chaque fichier contient plusieurs*

constructions du langage qui correspondent à des nœuds dans l'arbre syntaxique. Pour un fichier comportant x nœuds, un gène est représenté ainsi :$G=[N_1,\ldots,N_x]$.

L'opération de mutation peut être définie à partir de ce modèle de gène. Elle consiste à remplacer un nœud par un autre dans un gène. A partir de la grammaire du langage (décrite sous forme d'arbre d'héritage dans la Figure 24) il est possible de vérifier que le nouveau nœud est compatible avec l'ancien. L'opération de mutation génère ainsi un gène licite (un fichier source syntaxiquement correct pour le langage). La Figure 21 illustre un exemple pour l'opération de mutation sur un gène : un nœud de type `foreach` est remplacé par un nœud de type `while` dans la méthode `set`.

Opération de mutation pour le test système. *L'opération de mutation sélectionne un gène aléatoirement dans un individu et remplace un nœud de ce gène par un autre :*

$$G = [N_1, \ldots, N_i, \ldots, N_x] \Rightarrow G_{mut} = [N_1, \ldots, N_{imut}, \ldots, N_x]$$

```
using System;
namespace Id_1 {
  using System;
  protected class Id_2 {
    [AnAttribute1; AnAttribute2]
    public string aField;

    public ~Id_2() {} //~Id_2

    [AnAttribute1; AnAttribute2]
    public Id_2() {} //Id_2

    [AnAttribute]
    public virtual returnType aMethod (Type1 param1,
                Type2 param2) ;

    [AnAttribute]
    static Type aProperty {
      get {}
      set {
        aVariable = aValue + 3;
        for (int i=0 ; !Id_6||Id_8!=Id_3 ; i++)
        {foreach (nodes n in the_tree)
            {anObject.aMethod (param3, param4 );}}
      }
    }
    public returnType1 aMethod2 (Type3 param5) {} //aMethod2
  } //Id_2
}
```

```
using System;
namespace Id_1 {
  using System;
  protected class Id_2 {
    [AnAttribute1; AnAttribute2]
    public string aField;

    public ~Id_2() {} //~Id_2

    [AnAttribute1; AnAttribute2]
    public Id_2() {} //Id_2

    [AnAttribute]
    public virtual returnType aMethod (Type1 param1,
                Type2 param2) ;

    [AnAttribute]
    static Type aProperty {
      get {}
      set {
        aVariable = aValue + 3;
        for (int i=0 ; !Id_6||Id_8!=Id_3 ; i++)
        { while(cond1){
              aVariable1++;}}
      }
    }
    public returnType1 aMethod2 (Type3 param5) {} //aMethod2
  } //Id_2
}
```

Figure 21 – Exemple de mutation d'un gène pour lé génération de cas de test système

Les définitions du gène et de l'opération de mutation données ici, correspondent aux concepts de CASTESTSYSTEME et MUTATIONSYSTEME du diagramme de la Figure 17.

3.3 Etudes de cas pour un algorithme génétique

Pour étudier l'automatisation de l'optimisation de cas de test en utilisant un algorithme génétique, deux études de cas ont été faites, et ont été choisies pour être représentatives de langages (Eiffel, C#), de styles de programmes (pratiquement sans état interne en Eiffel) et de niveaux de test différents (unitaire, système). Avec la première, nous nous sommes intéressés à la génération de cas de test unitaires avec des classes écrites en Eiffel. Pour la seconde nous avons appliqué un algorithme génétique pour améliorer des cas de test pour système écrit en C# dans le cadre de la plate-forme .NET. Chaque étude de cas représente une catégorie particulière de logiciel. Les classes Eiffel, étudiées au niveau unitaire, manipulent peu d'attributs et ont de petites méthodes. Quant au système, il représente n'importe quel logiciel qui transforme des données en entrée dans un autre format de sortie. Par exemple, la même modélisation pour l'algorithme génétique peut être utilisée pour tester des logiciels fondés sur XML comme format d'échange.

3.3.1 Optimisation de cas de test unitaires : un exemple en Eiffel

L'étude de cas pour le test unitaire est fondée sur la librairie Pylon qui offre plusieurs structures de données pour le langage Eiffel. Cette librairie est disponible à l'adresse suivante : http://www.eiffel-forum.org/archive/arnaud/pylon.htm. Pour les expériences d'optimisation automatique de cas de test, nous avons choisi le paquetage de la librairie qui concerne la gestion du temps et des dates. La classe principale de ce paquetage est P_DATE_TIME, et une description complète du paquetage est donnée Figure 22.

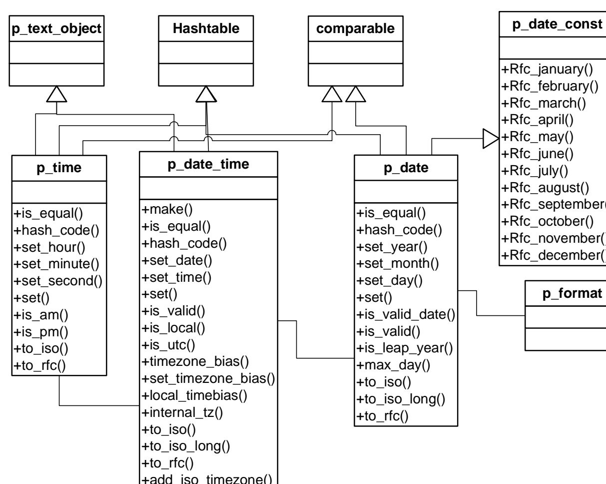


Figure 22 – Classes du paquetage « date-time » de la bibliothèque Pylon

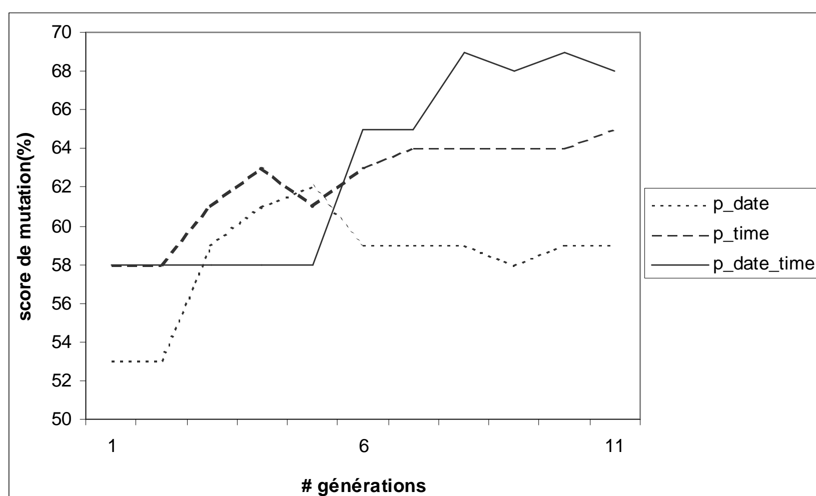


Figure 23 – Amélioration automatique du score de mutation pour 3 ensembles de cas de test

L'ensemble initial de cas de test pour chacune des classes P_TIME, P_DATE_TIME et P_DATE a été écrit par un groupe d'étudiants. Le score de mutation pour chaque ensemble de cas de test a alors été calculé. Les résultats ainsi que le nombre de mutants générés pour calculer ces scores sont donnés dans le Tableau 2. Le nombre de mutants varie d'une classe à l'autre en fonction du nombre de méthodes dans les classes et de la complexité de ces méthodes. Plus il y a de code dans une classe, plus on peut insérer d'erreurs. De la même façon, plus le code est complexe (points de contrôle, prédicats dans ces points de contrôle, nombre de paramètres…), plus le nombre de mutants générés est grand. Par exemple, il y a moins de mutants pour la classe P_DATE_TIME car la plupart de ces méthodes se contentent de déléguer leur traitement à des méthodes des classes P_DATE ou P_TIME. Ces méthodes sont donc très simples, et peu d'erreurs peuvent y être injectées.

La suite de l'expérience consiste à améliorer automatiquement ces ensembles de cas de test initiaux grâce à un algorithme génétique. L'algorithme a été exécuté pour chaque ensemble. Chaque ensemble initial contenait trois cas de test qui ont servi de gènes pour construire la population initiale de l'algorithme. Dans les trois cas, la population initiale était de 15 individus de taille 5. Pour les résultats présentés Figure 23, le taux de mutation était amélioré de 10%.

Tableau 2 – Scores de mutation des ensembles initiaux de cas de test pour le paquetage « date-time »

	p_date	p_date_time	p_time
# mutants générés	673	199	275
score de mutation (%)	53	58	58

3.3.2 Optimisation de cas de test système : l'exemple d'un composant .NET

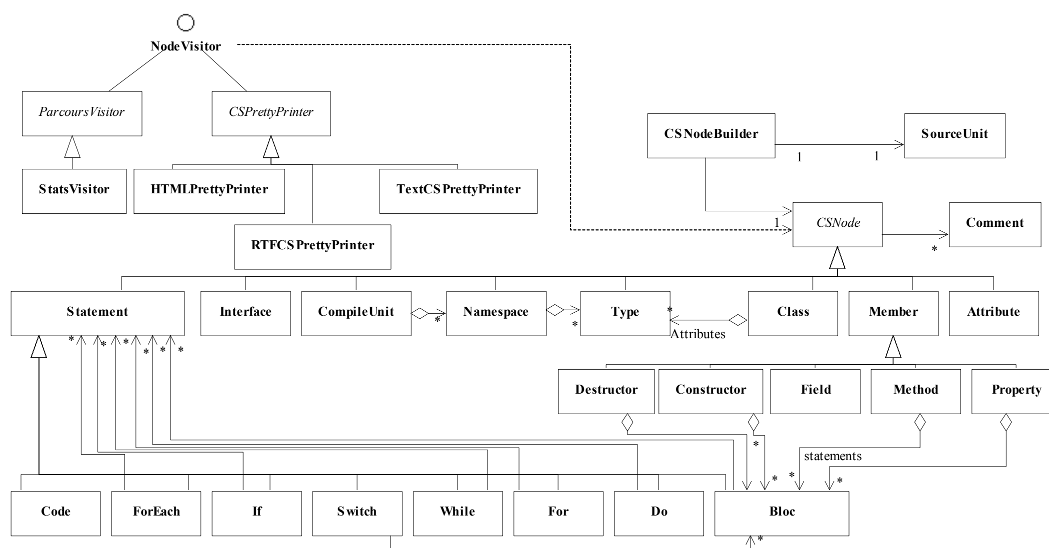


Figure 24 – Parseur pour le langage C#

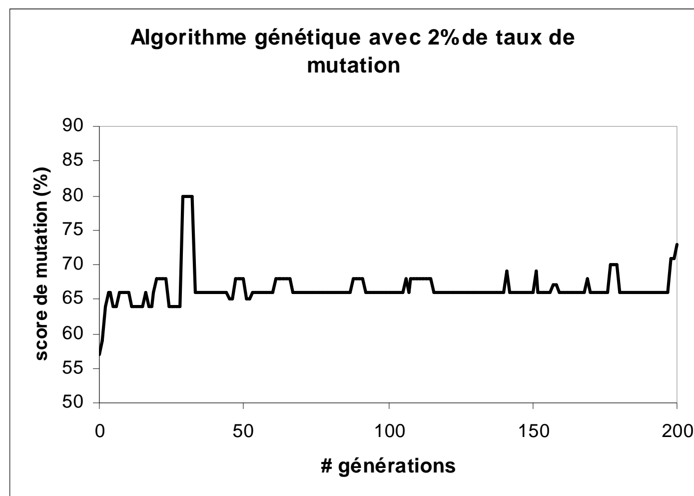
Nous avons expérimenté le modèle d'algorithme génétique pour l'amélioration de cas de test système sur un composant .NET écrit en C#. Ce composant est un parseur pour le langage C# [MSDN"02a; MSDN"02b] dont le diagramme de classes est donné Figure 24. Ce diagramme est composé de 32 classes qui peuvent être divisées en trois sous-ensembles principaux. Tout d'abord, la classe CSNODEBUILDER est la classe principale pour la construction de l'arbre syntaxique, et la hiérarchie d'héritage à partir de la classe CSNODE représente les différents types de nœuds possibles dans l'arbre. Enfin, les classes qui implémentent l'interface NODEVISITOR constituent le troisième sous-ensemble. Ces classes correspondent à l'application du design pattern visitor [Gamma"95] et représentent différents traitements possibles sur l'arbre syntaxique. Par exemple, la classe TEXTCSPRETTYPRINTER implante l'impression de l'arbre syntaxique dans un fichier au format texte. Ce composant prend un ou plusieurs fichiers source C# en entrée, et construit l'arbre syntaxique correspondant.

Pour l'étude de l'algorithme génétique sur ce système, nous avons généré 500 mutants en nous restreignant à l'opérateur NOR (cf. section 3.1.4). Les résultats obtenus sont néanmoins intéressants puisque le score de mutation ainsi obtenu correspond au taux de couverture des instructions des cas de test générés. La plupart des mutants ont été générés sur les classes TEXTCSPRETTYPRINTER, TOKENIZER et CSNODEBUILDER qui exécutent les opérations les plus complexes du système. La population initiale est constituée de 12 individus de taille 4, et le score de mutation initial est de 56%. Les résultats pour l'amélioration automatique du score avec un algorithme génétique sont donnés Figure 25.

3.3.3 Résultats et remise en cause du modèle génétique

Cette section résume plusieurs conclusions à propos de l'application d'un algorithme génétique pour l'optimisation automatique de cas de test (cf. Figure 23 et Figure 25). Nous nous intéressons d'abord aux intérêts de l'approche pour améliorer la qualité des classes/composants sous test. Puis, nous expliquons le manque d'efficacité de ce modèle pour résoudre notre problème. Quelques points particuliers sont examinés en détail : la croissance lente et irrégulière du score de mutation, le coût en termes de temps d'exécution et les problèmes de calibrage du modèle.

Pour les deux études de cas, l'algorithme génétique a automatiquement amélioré la qualité des ensembles de cas de test initiaux. Ces ensembles optimisés ont ensuite été exécutés sur les classes/composants sous test. Plusieurs erreurs ont ainsi été détectées puis corrigées. Au cours du diagnostic sur les mutants vivants, nous avons repéré des mutants équivalents et du « code mort ». En effet, si les erreurs sont injectées dans des parties du code qui ne sont jamais exécutées, *i.e.*, du « code mort », le mutant ne peut pas être tué. Malgré cette amélioration de la qualité du composant sous test grâce à l'algorithme génétique, les résultats des expériences sont décevants, tant pour la génération de cas de test unitaires (expérimentations sur les classes de gestion de date) que pour les cas de test systèmes (pour le parseur C#). En effet, dans les deux cas le score de mutation n'augmente pas de manière significative (il ne dépasse jamais 90%) et le processus ne converge pas (pas de croissance stricte du score). Enfin, nous avons eu recours à des taux inhabituels pour les opérateurs de croisement et de mutation pour améliorer l'efficacité de l'approche, ce qui est suffisant pour remettre en cause le modèle génétique pour notre problème. Dans la suite de cette section nous revenons sur ces différents points et proposons des solutions pour améliorer le modèle initial.



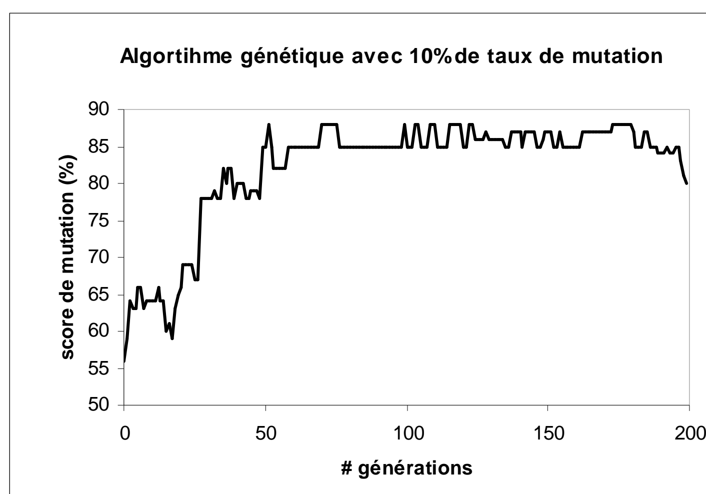


Figure 25 – Algorithme génétique pour l'optimisation automatique de cas test pour un parseur C#

Pour rendre compte de la non-croissance stricte du score de mutation il suffit de remarquer qu'au cours du passage d'une génération à la suivante, certains individus sont sélectionnés en fonction de leur valeur d'utilité, puis ils sont reproduits, croisés et certains gènes sont mutés pour obtenir une nouvelle population. Certaines données de test utiles peuvent donc être éliminées car non-sélectionnées pour la reproduction, et de même la mutation peut détruire une donnée de test présente uniquement dans le gène muté. Le passage d'une génération peut donc entraîner une perte d'information, et le meilleur individu de la nouvelle génération peut être moins bon que celui de la génération précédente. Cette perte d'information entraîne une amélioration lente, et même des chutes dans la courbe de l'évolution du score de mutation. La mémorisation des individus sélectionnés pour la reproduction éviterait ces problèmes. En effet, mémoriser certains individus permet au score de mutation de ne pas descendre en dessous de la valeur atteinte par les individus mémorisés.

La seconde limite du modèle est liée aux coûts de paramétrage du modèle. Un algorithme génétique cherche l'individu qui a la valeur d'utilité optimale, et n'essaie pas d'optimiser la valeur optimale globale d'une population. La taille des individus doit donc être suffisamment grande, dés l'initialisation de l'algorithme, pour contenir assez de gènes (cas de test) pour atteindre la valeur d'utilité optimale. Or, il est très difficile de prévoir combien de cas de test seront nécessaires pour tuer tous les mutants d'un composant particulier. Il faut donc commencer avec des tailles très grandes, puis paramétrer le modèle pour que le taille choisie soit suffisamment grande pour atteindre le score optimal, et cependant pas trop grande pour que l'individu généré (un ensemble de cas de test) ne soit pas trop long à exécuter et à comprendre. De plus, le paramétrage du modèle doit être refait pour chaque nouveau composant sous test. Même si cet effort de paramétrage est une contrainte pour chaque application d'un algorithme génétique, il semble particulièrement contraignant dans notre cas puisque la taille de l'ensemble solution n'est pas importante. Une adaptation du modèle

consisterait à ne plus contraindre la taille de l'ensemble de cas de test qu'on essaie d'améliorer. La taille de l'individu est donc particulièrement difficile à estimer, et la notion même d'individu (qui vise à générer « un bon ensemble ») apparaît inadaptée.

Le second paramètre qu'il faut fixer est le taux de mutation à chaque génération. Pour chacune des expériences, il a fallu fixer ce taux à une valeur largement supérieure à celle utilisée pour une application normale d'un algorithme génétique. La Figure 25 montre des résultats pour des taux de mutation de 2% et 10%. Pour le taux de mutation le plus faible (qui correspond au taux habituellement utilisé pour un algorithme génétique), l'optimisation automatique ne donne aucun résultat : le score de mutation ne dépasse jamais 80% et ne se stabilise jamais. Pour le taux de mutation de 10% les résultats sont meilleurs, puisque le score de mutation augmente jusqu'à atteindre des pics à 90%. Ce score n'est jamais stabilisé pour autant. Par ailleurs, il apparaît que l'opération de mutation est celle qui crée le plus d'information puisqu'elle génère de nouvelles données de test, ce qui permet de générer des cas de test qui couvrent de nouvelles parties du composant sous test. Ceci explique pourquoi un taux de mutation élevé donne de meilleurs résultats, mais est en contradiction avec l'application normale des algorithmes génétiques pour lesquels l'opération de mutation est de faible importance et donc rarement utilisée.

Enfin, la troisième limite du modèle concerne l'opérateur de croisement. Le problème avec cet opérateur est plus son manque d'efficacité pour résoudre notre problème que son paramétrage. En effet, le modèle de gène est tel que chaque gène peut être exécuté avec le composant sous test, indépendamment de tout individu. Les gènes sont indépendants les uns des autres, et l'ordre dans lequel ils sont exécutés n'a pas d'impact sur le score de l'individu, ce qui démontre que la notion d'individu comme « séquence de gènes » n'est pas exploitée pour notre problème. Il apparaît donc que l'opérateur de croisement n'est pas efficace dans notre contexte, puisque sa fonction est de créer de l'information en réordonnant les gènes à l'intérieur des individus.

Pour conclure cette discussion, il semble que le modèle d'algorithme génétique ne soit pas adapté à l'optimisation automatique de cas de test. Une adaptation de ce modèle devrait prévoir une mémoire et supprimer la notion d'individu et ne garder que des gènes (cas de test). Ceci éviterait les problèmes de paramétrage pour chaque composant sous test et permettrait une croissance stable du score de mutation. Cependant, ces expériences ont montré certains aspects intéressants de ce modèle. Tout d'abord le modèle de gène est clairement défini et correspond bien à ce qu'il faut optimiser : on peut le réutiliser. Ensuite, l'opération de mutation semble être bien modélisée aussi pour introduire de l'information. Enfin, le score de mutation est un bon indicateur de l'utilité d'un gène pour guider la génération vers une bonne solution. La section suivante décrit un nouveau modèle prenant en compte les différentes remarques énoncées ici pour adapter l'algorithme génétique. Ce modèle est appelé « approche bactériologique » et est fondé sur le phénomène d'adaptation bactériologique [Rosenzweig"95].

3.4 Une approche adaptative : un « algorithme bactériologique »

Les expériences décrites dans la section 3.3 ont montré les limites de l'application d'un algorithme génétique pour l'optimisation automatique de cas de test. Cette section présente une adaptation du modèle génétique pour notre problème particulier. L'adaptation essentielle consiste à mémoriser les meilleurs cas de test d'une génération à l'autre. Dans ce cas, il est possible de retirer les mutants tués par ces cas de test de l'ensemble des mutants à tuer. Ceci a pour conséquence de diminuer le temps de calcul pour une génération lorsque l'ensemble des mutants vivants diminue.

Même si l'adaptation du modèle génétique semble fondée sur de petits changements, elle transforme complètement l'idée de l'algorithme. Un algorithme génétique étudie l'ensemble des solutions possibles à un problème, à la recherche de l'individu optimal. A l'inverse, pour le nouveau processus, l'ensemble des solutions peut changer d'une génération à l'autre puisque le but (tuer tous les mutants encore vivants) peut changer d'une génération à l'autre. De plus, le nouveau processus ne génère pas l'individu optimal, mais un ensemble de bons individus (ceux qui ont été mémorisés au cours du processus de génération). Notre nouvelle approche est donc éloignée du modèle génétique dans ses principes. Cependant, elle correspond toujours à une analogie avec un phénomène biologique appelé « adaptation bactériologique » [Rosenzweig"95].

3.4.1 Le modèle bactériologique

a Le processus global

Le but de cet algorithme est de générer un ensemble de cas de test efficace et de taille raisonnable pour un programme particulier. L'ensemble solution est composé d'un certain nombre d'éléments appelés *bactéries*, qui correspondent aux unités atomiques du nouveau modèle. L'algorithme prend un ensemble initial de bactéries en entrée et son évolution consiste en une série de mutations successives sur les bactéries pour explorer l'ensemble des solutions possibles. L'ensemble solution est construit de manière incrémentale en ajoutant des bactéries qui peuvent améliorer la qualité de l'ensemble. Au cours de l'exécution on distingue donc deux ensembles de bactéries, l'ensemble solution (en construction) et l'ensemble des bactéries potentielles.

Comme les bactéries sont les seules éléments manipulés par cet algorithme, et que ce sont des unités atomiques, l'opérateur de croisement n'est plus présent dans ce nouveau processus de génération. Il n'y a plus que deux opérations à définir pour l'algorithme bactériologique : la *fonction d'utilité* et l'*opérateur de mutation*. Nous reprenons les mêmes opérateurs de mutation que dans le cas de l'algorithme génétique (changement des paramètres d'appel dans le cas unitaire, et changement d'un nœud syntaxique dans le cas système). Pour la fonction d'utilité, on distingue entre l'utilité d'un ensemble de bactéries, et l'utilité d'une bactérie qui

correspond à la capacité d'une bactérie à augmenter la qualité d'un ensemble. La fonction d'utilité pour un ensemble de bactéries est la même que pour l'algorithme génétique (puisqu'un individu de l'algorithme génétique est un ensemble de gènes). La fonction d'utilité pour une bactérie s'exprime à partir de la fonction sur un ensemble.

Fonction d'utilité pour une bactérie. *Soit S un ensemble de bactéries et F la fonction d'utilité pour un ensemble de bactéries. La valeur d'utilité f(b) pour une bactérie b, par rapport à S, est calculée de la manière suivante : $f(b) = F(S \cup b)-F(S)$. Plus la bactérie peut apporter de l'information à l'ensemble, plus sa valeur d'utilité est grande.*

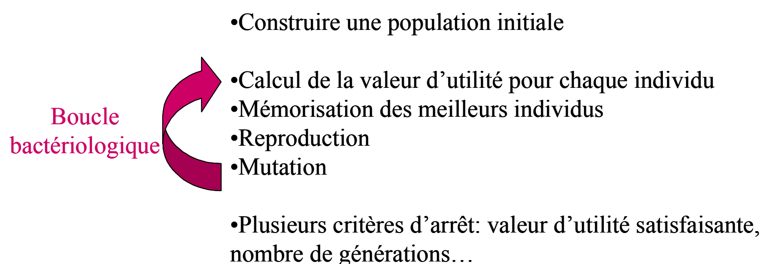


Figure 26 – Le processus bactériologique

L'algorithme évolue de manière incrémentale grâce à ces différents opérateurs. L'algorithme calcule la valeur d'utilité de chaque bactérie à chaque étape. La bactérie qui améliore le plus la qualité de l'ensemble solution est sélectionnée et ajoutée à l'ensemble. Puis les bactéries qui ont une valeur d'utilité nulle sont supprimées de l'ensemble des bactéries potentielles. L'algorithme sélectionne ensuite des bactéries pour la reproduction de la même manière que pour l'algorithme génétique, et applique l'opérateur de mutation sur chaque bactérie sélectionnée. Le processus général est décrit Figure 26 et le texte de l'algorithme est détaillé dans la section suivante.

b le texte de l'algorithme

Pour la description du texte de l'algorithme bactériologique, on suppose l'existence de la fonction `select`. L'implantation des fonctions `Utilité` et `Mute` dépend de l'application particulière de l'algorithme génétique. Toutes ces fonctions sont décrites brièvement avant l'algorithme bactériologique.

```
// La fonction select prend un ensemble de bactéries et une séquence de réels
en entrée. La séquence de réels doit être de la même taille que l'ensemble de
bactéries. La fonction rend un individu tiré au hasard dans le bouillon, une
bactérie i ayant une probabilité proba[i] d'être tirée.
select(Ens[Bactérie] bouillon, Ens[Réel] proba) : Bactérie

//La fonction Utilité rend la valeur d'utilité d'un bouillon bactériologique
Utilité (Ens[Bactérie] bouillon) : Réel

//La fonction Mute rend la bactérie bact mutée
Mute (Bactérie bact) : Bactérie
```

```
//l'algorithme bactériologique utilise une variable globale pour sauvegarder
les bactéries au cours de l'exécution
Ens[Bactérie] solution ← ∅ ;

//l'algorithme bactériologique prend en entrée : un bouillon initial, un
objectif, un taux de mutation, un seuil de mémorisation et nombre de bactéries
à générer à chaque génération
Algo_bactériologique (Ens[Bactérie] bouillon,
                    Réel obj,
                    Réel tx_mutation,
                    Réel seuil_mémo,
                    Entier nb_bact)
  début
    tantque ( ! Utilité(solution) > obj) faire
      solution ← solution ∪ select_meilleure(bouillon) ;
      bouillon ← bouillon \ {b | Utilité_relative(b) <= 0} ;
      bouillon ← bouillon ∪ générer(nb_bact, bouillon) ;
    fait
  fin

select_meilleure (Ens[Bactérie] bouillon, Réel seuil_mémo, Réel obj): Bactérie
  local Bactérie mb, b ;
  début
    mb ← b ∈ bouillon | Utilité_relative(b) = max_{x ∈ bouillon}(Utilité_relative(x))
    si (Utilité(solution ∪ mb) / obj) > seuil_mémo alors Result ← mb
    sinon Result ← ∅
  fin

générer(Entier nb_bact, Ens[Bactérie] bouillon) : Ens[Bactérie]
  localRéel u_cumulée ;
      Ens[Réel] proba ;
  début
    u_cumulée ← Σ_{i=0}^{bouillon.taille} Utilité_relative(bouiilon[i]) ;
    ∀i ∈ [0..bouillon.taille-1], proba[i] ← pop[i]*(Utilité(pop[i])/u_cumulée)
    pour i de 0 à nb_bact-1 faire
      bouillon ← bouillon ∪ Mute(select(bouillon, proba)) ;
    fait
    Result ← bouillon ;
  fin

//La fonction Utilité_relative rand la valeur d'utilité d'une bactérie
relative à l'ensemble de bactéries solution
Utilité_relative (Bactérie bact) : Réel
  début
    Result ← Utilité(solution ∪ bact) – Utilité(solution)
  fin
```

c Le modèle pour l'optimisation de cas de test

La Figure 27 illustre l'architecture générale du modèle d'adaptation bactériologique. Cette approche est un nouveau modèle pour l'optimisation de cas de test, la classe BACTERIOLOGIQUE hérite donc de TESTOPTIMISEUR. Une bactérie est modélisée comme un cas de test (CASTEST implante l'interface BACTERIE), et les deux classes qui spécialisent CASTEST représentent les types de cas de test aux niveaux unitaire et système. Les cas de test

pour le modèle bactériologique sont les mêmes que dans le cadre génétique. L'opérateur de mutation est présent aussi dans le cas de l'algorithme bactériologique. Puisque la structure des bactéries est la même que celle des gènes, l'opérateur de mutation ne change pas non plus (MUTATIONUNITAIRE et MUTATIONSYSTEME).

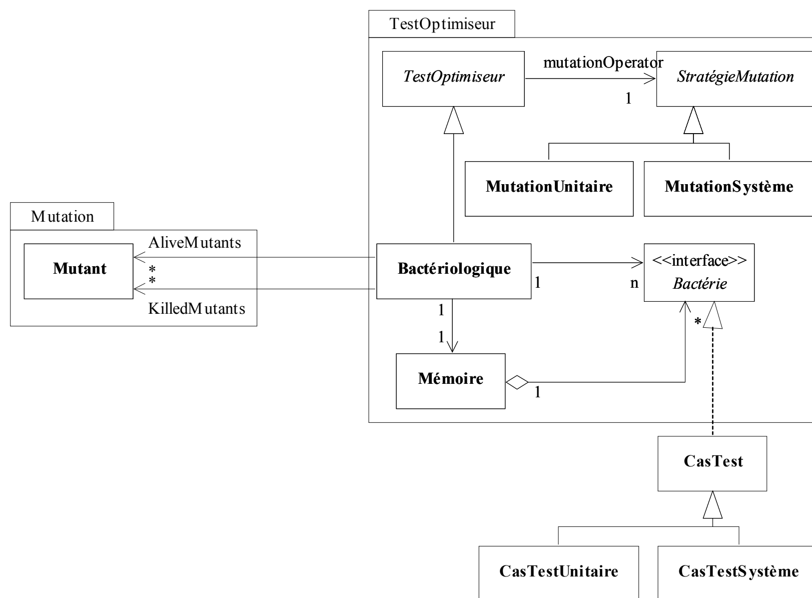


Figure 27 – Architecture générique de l'approche bactériologique pour l'optimisation de cas de test

Par ailleurs, puisque l'algorithme bactériologique ne manipule que des bactéries qui correspondent à des unités pour le modèle, la notion d'individu ainsi que les opérateurs de reproduction et croisement ont disparu. L'abandon de l'opérateur de croisement est une différence majeure avec le modèle génétique. Ceci correspond à une évolution qui nous semblait nécessaire au regard des résultats de l'algorithme génétique, puisque cet opérateur n'était pas efficace pour la convergence vers une solution optimale (cf. section 3.3.3).

Cette approche, comme la précédente, nécessite une *fonction d'utilité* pour sélectionner les meilleures bactéries qui sont mémorisées. Puisqu'une bactérie est modélisée par un cas de test, le score de mutation est utilisé comme valeur d'utilité pour un gène.

Enfin, deux autres différences apparaissent par rapport au modèle précédent : l'apparition d'une MEMOIRE, et deux associations vers la classe MUTANT au lieu d'une. L'algorithme bactériologique utilise une mémoire qui contient, à un instant donné, les meilleures bactéries produites au cours des générations précédentes. D'autre part, l'algorithme génétique calculait le score de mutation des cas de test avec tous les mutants à chaque génération, la classe GENETIQUE n'avait donc qu'une association vers la classe MUTANT représentant l'ensemble des mutants générés pour le composant sous test. Dans le cas de l'algorithme bactériologique, certaines bactéries sont sauvegardées au cours de l'exécution. On distingue alors les mutants

qui sont tués par les bactéries sauvegardées, des mutants qui n'ont pas encore été tués par une bactérie. Il y a donc deux associations de la classe BACTERIOLOGIQUE vers la classe MUTANT, représentant les ensembles de mutants tués et vivants.

3.4.2 De nouveaux résultats

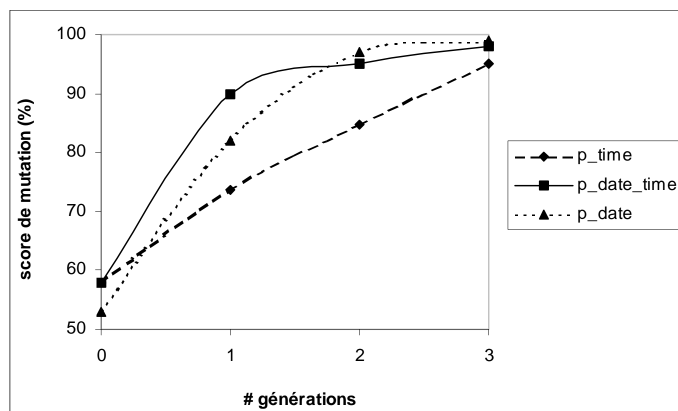


Figure 28 – Résultats de l'algorithme bactériologique pour l'amélioration automatique de cas de test unitaires

Les Figure 28 et Figure 29 donnent l'évolution du score de mutation pour les cas de test unitaires des trois classes *P_TIME*, P_DATE et P_DATE_TIME, ainsi que pour les cas de test du parseur C#. Le score de mutation de ces ensembles de cas de test a été amélioré automatiquement grâce à un algorithme bactériologique. Pour ces expérimentations, deux paramètres on dû être fixés : le nombre maximal de bactéries sauvegardées lors du passage d'une génération à l'autre, et la taille d'une bactérie. Puisque l'ensemble initial de bactéries était petit dans tous les cas (entre 3 et 10 bactéries), nous avons décidé de ne sauvegarder qu'une seule bactérie à la fin d'une génération. La taille d'une bactérie correspond à des paramètres différents en fonction du type de cas de test généré (cf. définitions suivantes). Une discussion sur le paramétrage de cette taille dans le cas de l'étude du parseur est présentée section 1.1.

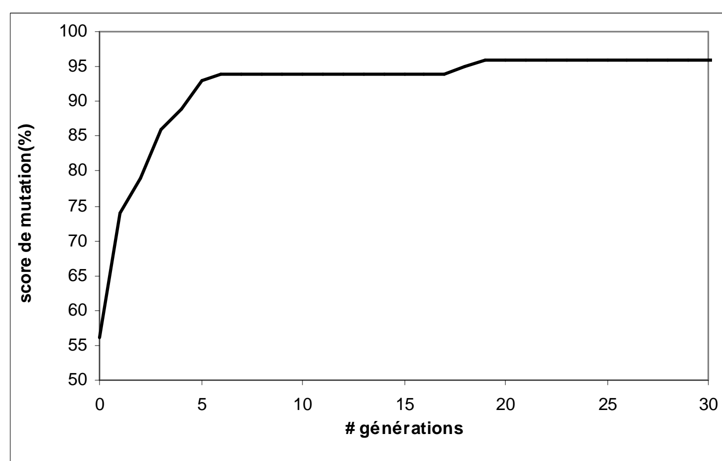


Figure 29 – Résultats de l'algorithme bactériologique pour l'amélioration automatique de cas de test systèmes

Taille d'un cas de test unitaire. *Soit B=[I,S] un cas de test unitaire pour lequel S est un ensemble d'appels de méthodes sur des instances de la classe sous test. La taille d'un tel cas de test est le cardinal de l'ensemble S.*

Taille d'un cas de test système. *Soit* $B=[N_1,\ldots,N_X]$ *un cas de test pour un parseur contenant X constructions du langage (X nœuds de l'arbre syntaxique). Alors le nombre X est aussi la taille de la bactérie modélisée par ce cas de test.*

3.4.3 Discussion et validation du modèle bactériologique

Cette section présente les gains en performance de l'algorithme bactériologique par rapport à l'algorithme génétique pour la génération de test. Nous discutons ensuite de différents problèmes qui peuvent apparaître en utilisant cet algorithme.

a Les performances

L'algorithme bacétriologique converge plus rapidement que le précédent. Le Tableau 3 résume les performances des deux approches pour le parseur C#. Cette table donne le nombre de générations nécessaire pour atteindre le score de mutation donné dans la deuxième colonne. L'algorithme bactériologique converge beaucoup plus rapidement que le génétique : 30 générations au lieu de 200. Cependant, comme les calculs effectués au cours d'une génération ne sont pas les même dans les deux approches, la dernière colonne du tableau donne des chiffres comparables : le nombre de programmes mutants exécutés pour atteindre le score de mutation maximal. Ces chiffres correspondent à une meilleure estimation que le nombre de générations de la complexité de chaque approche. En effet l'exécution d'un mutant avec un cas de test est la tâche la plus coûteuse dans les deux cas.

Tableau 3 – Comparaison des performances des algorithmes

Algorithme	# générations	score (%)	# mutants exécutés
Génétique	200	85	480000
Bactériologique	30	96	46375

Ces nouvelles expériences ont donné d'autres résultats intéressants. Tout d'abord, la mémorisation des meilleures bactéries évite les creux dans l'évolution du score de mutation, rendant l'algorithme bactériologique plus rapide que le précédent. Ensuite, l'approche bactériologique est plus facile à paramétrer grâce à la suppression de plusieurs paramètres (taille des individus, et taux de croisement parmi ces individus). Ceci rend cette approche plus facile à réutiliser pour l'optimisation automatique de cas de test. L'algorithme est aussi davantage contrôlable puisque l'évolution de l'algorithme est moins aléatoire.

b Limites de l'algorithme

Deux inconvénients apparaissent en passant de l'algorithme génétique à l'approche bactériologique. Premièrement, la possible perte d'information due au fait que les bactéries d'une nouvelle génération sont générées à partir des meilleures bactéries uniquement. En effet, certaines informations peuvent n'être présentes que dans les bactéries moins efficaces. Pour résoudre cette perte d'information potentielle, une solution consiste à tirer au hasard une bactérie sauvegardée pour générer une nouvelle population si le score stagne trop longtemps.

Ensuite, l'ensemble final de cas de test peut ne pas être minimum en appliquant l'algorithme bactériologique. Par exemple, neuf cas de test ont été générés dans le cas du parseur C# au lieu de quatre avec l'algorithme génétique. Or, il est important d'avoir un ensemble de cas de test de taille raisonnable pour le test de non-régression, puis lors de la maintenance du système.

Deux techniques permettraient de résoudre ce problème : minimiser l'ensemble de cas de test générés à la fin de l'algorithme ou sauvegarder moins de bactéries. Il est possible de sauvegarder moins de bactéries en atteignant le même score de mutation final. Pour cela, il faut fixer un seuil au-delà duquel les bactéries sont sauvegardées, ce qui force l'algorithme à générer plus de nouvelle information avant la mémorisation. L'ensemble final atteint donc des scores de mutation similaires avec moins de bactéries. Des expériences avec différents seuils pour la sauvegarde sont détaillées dans la section 1.1.

Le problème de la minimisation d'un ensemble de cas de test, consiste à trouver un sous-ensemble de cas de test assurant un pouvoir de détection d'erreur équivalent à celui de l'ensemble initial. Dans le cas particulier de l'analyse de mutation, ce problème peut se ramener à la minimisation de la matrice couverture des mutants par les cas de test. Cette matrice s'obtient en calculant les signatures complètes de toutes les bactéries sur tous les mutants. Pour k bactéries et n mutants, on obtient une table T de booléens de taille (n+1)*k telle que:

$$\forall i \in [1,n], \forall j \in [1,k],\ T(B_i, M_j) = \begin{cases} 1 & \text{si la bactérie } B_i \text{ tue mutant } M_j \\ 0 & \text{sinon} \end{cases}$$

$T(n+1, M_j) = \bigcup_{i=1}^{n} T(B_i, M_j)$, c'est-à-dire que $T(n+1\ , M_j)$ vaut 1 si le mutant M_j est tué par une bactérie. Cette dernière ligne correspond donc à la signature globale de l'ensemble de n bactéries. La Figure 30 donne un exemple de matrice obtenue. La minimisation de la matrice permet d'obtenir l'ensemble de cas de test minimal nécessaire pour tuer tous les mutants.

	M1	M2	M3		Mk-1	Mk
B1	1	1	0		0	0
B2	0	0	1		0	0
B3	0	0	1		1	0
•						
•				• • • •		
Bn-1	1	0	1		0	0
Bn	0	0	0		1	0
	1	1	1		1	0

Figure 30 – Exemple de table pour le calcul de l'ensemble minimal de tests

Les nouveaux résultats montrent que les adaptations de l'algorithme génétique qui avaient été détectées comme nécessaires au cours de la discussion de la section 3.3.3 sont de bonnes heuristiques pour résoudre notre problème. L'application de ces heuristiques nous a conduits à définir un nouveau modèle appelé *modèle bactériologique*. Cet algorithme apparaît plus souple et stable que l'algorithme génétique. La section suivante détaille les expériences conduites sur le parseur C# pour le paramétrage du paramètre principal du modèle bactériologique, à savoir la taille d'une bactérie. Dans un second temps, et pour que l'étude expérimentale soit aussi exhaustive que possible, nous avons cherché s'il existait un algorithme intermédiaire plus efficace entre l'algorithme génétique sans mémorisation et l'algorithme bactériologique qui mémorise toute bactérie améliorant le score de mutation de l'ensemble.

3.5 Paramétrage des modèles

3.5.1 Paramétrage du modèle bactériologique

L'unique paramètre qu'il faut fixer pour l'algorithme bactériologique est la taille des bactéries. Dans le cas du test unitaire, cette taille est définie par le nombre d'appels de méthode inclus dans un cas de test (cf. section 3.4.2). Pour le test système, la taille d'une bactérie dépend du format des données d'entrée pour le programme sous test. Dans le cas du parseur C#, la taille est donnée par le nombre de nœuds de l'arbre syntaxique contenu dans un cas de test. Cette modélisation de la taille d'une bactérie pourrait s'appliquer à tout logiciel qui transforme des données d'un format à un autre, notamment des logiciels fondés sur XML. La Figure 31, donne la courbe en trois dimensions et sa projection plane (des niveaux de gris exprimant les variations du score de mutation) pour la relation entre la taille de la bactérie, le score de mutation et le nombre de générations. Notons que cette courbe garde la même allure, même en répétant les expérimentations (d'où la ligne médiane en noir sur la figure). Les conclusions sont les suivantes :

- si la bactérie est trop petite (<5), l'algorithme ne peut pas atteindre un score de mutation satisfaisant. Ceci s'explique par le fait qu'une bactérie est un cas de test, et que si un cas de test est trop petit, il ne peut pas exécuter de scénario complexe, et s'avère donc incapable d'atteindre certaines parties du code. La taille de 5 signifie

que les cas de test doivent comporter plus de 5 appels de méthodes pour couvrir tout le code unitaire ou plus de 5 nœuds syntaxiques dans le cas du test système.

- entre 5 et 15 la vitesse de convergence augmente.
- lorsque la taille de la bactérie dépasse 15, la vitesse de convergence se stabilise.

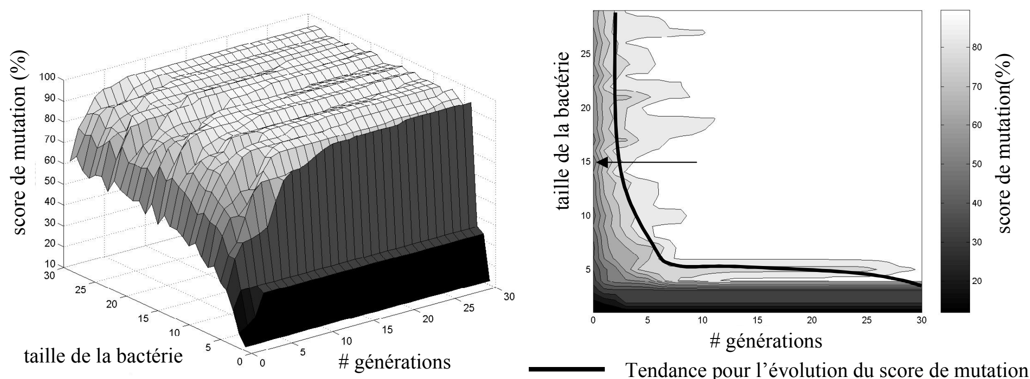


Figure 31 – Paramétrage de la taille de la bactérie

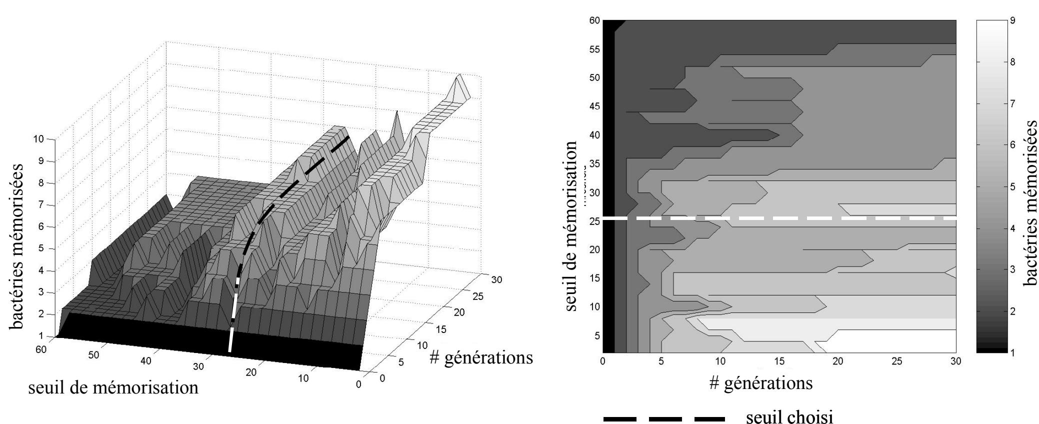


Figure 32 – Paramétrage du seuil de mémorisation pour le nombre de bactéries mémorisées

Nous avons besoin de cas de test de taille minimum, pour limiter les temps d'exécution, éviter les données de test redondantes et faciliter l'interprétation des données de test pour le testeur. Pour les expériences des sections 3.3.2 et 3.4.2 et , nous avons choisi 15 comme taille d'une bactérie pour les cas de test système.

3.5.2 Recherche d'un algorithme intermédiaire

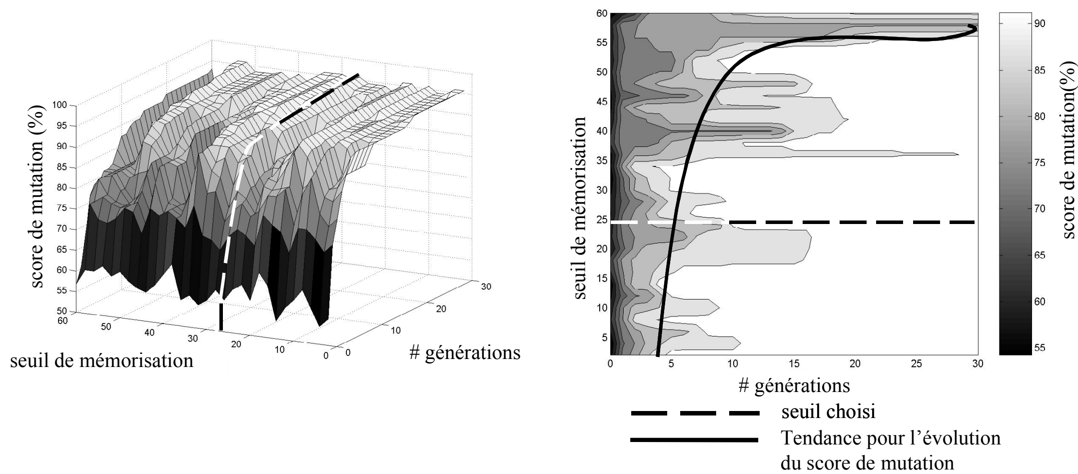


Figure 33 – Paramétrage du seuil de mémorisation pour le score de mutation

Cette section aborde une approche intermédiaire entre les algorithmes génétique et bactériologique. Le but des expériences présentées ici, est de déterminer l'existence d'un optimum local meilleur que les deux solutions explorées jusqu'à présent. Cette approche mixte a été paramétrée grâce à des expériences répétées sur le parseur C#.

La différence essentielle entre l'algorithme génétique de l'algorithme bactériologique tient à la notion de mémorisation. Dans le cas de l'algorithme génétique, aucun cas de test n'est mémorisé ; à l'inverse de l'algorithme bactériologique pour lequel une bactérie est systématiquement mémorisée si elle apporte de l'information. Entre ces deux solutions opposées, nous fixons un seuil minimum pour la valeur d'utilité de la bactérie sauvegardée. Les trois approches sont ainsi résumées :

```
Soit B une bactérie
  si val_utilité(B)> val_seuil alors sauvegarde B
```

Le type de l'algorithme dépend de la valeur de seuil:

```
  si val_seuil = 100 alors "pur " génétique
  si val_seuil = 0 alors "pur" bactériologique
  si  0 < val_seuil < 100 alors approche mixte
```

Deux critères sont pris en compte pour déterminer la meilleure valeur pour le seuil de mémorisation:

- minimisation du nombre de bactéries sauvegardées à la fin du processus, pour minimiser le temps d'exécution et la difficulté d'interprétation des cas de test. La Figure 32 montre l'impact de la valeur de seuil sur le nombre de bactéries mémorisées.
- minimisation du nombre de générations pour atteindre le score de mutation optimal. La Figure 33 présente l'impact de la valeur de seuil sur la vitesse de convergence.

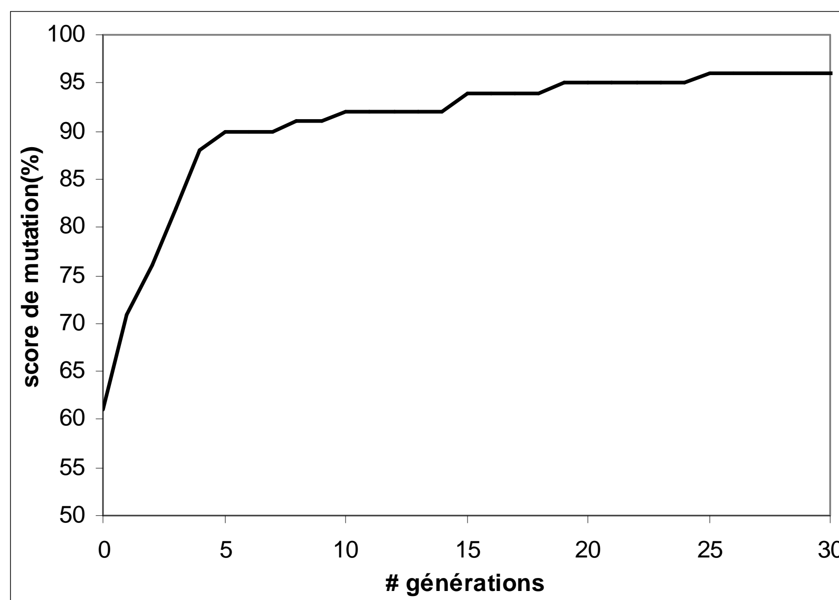


Figure 34 – Évolution du score de mutation avec une approche mixte

La Figure 32 montre que le nombre de bactéries sauvegardées décroît régulièrement avec l'augmentation de la valeur du seuil de mémorisation, la Figure 33 indique que la vitesse de convergence augmente lorsque le seuil dépasse 30%. Un compromis peut être trouvé pour minimiser ces deux valeurs lorsque le seuil se situe entre 20 et 30%. Dans ce cas, la vitesse de convergence est satisfaisante (autour de 30 générations pour atteindre le score optimal) et l'ensemble final de bactéries contient 7 bactéries. La Figure 34 montre les résultats obtenus pour une valeur de seuil de 25%.

En conclusion, l'approche mixte permet de réduire la taille de l'ensemble final de cas de test de 10 à 7. Par contre, le score de mutation de 95% n'est atteint qu'après 25 générations au lieu de 20 dans le cas de l'algorithme bactériologique (cf. Figure 29). Le gain de cette approche mixte nous semble donc peu significatif par rapport à l'effort de paramétrage qu'il nécessite. En effet, de nombreuses expérimentations sont nécessaires pour trouver un bon compromis pour la valeur de seuil de mémorisation. A l'inverse, l'approche bactériologique semble plus facile à réutiliser et généraliser.

3.6 Conclusion

L'efficacité des cas de test pour détecter des erreurs dans un composant est essentielle pour la qualité du composant. La méthode de conception, proposée au chapitre précédent, prévoit d'évaluer l'efficacité des cas de test pour un composant grâce à l'analyse de mutation. Notre expérience montre que les cas de test fournis par le testeur atteignent un score de mutation entre 50 et 70%, mais qu'améliorer ce score pour dépasser 90% est très coûteux en temps et en effort. Dans ce chapitre, nous avons donc étudié l'amélioration automatique d'un ensemble initial de cas de test.

La complexité de ce problème d'optimisation nous a tout d'abord amenés à utiliser un algorithme génétique pour le résoudre. Cependant, dés la modélisation du problème, nous avons dû adapter certains aspects de l'algorithme (taux de mutation, modélisation du gène) de manière tellement forte que l'algorithme s'éloignait d'un algorithme génétique classique. Ensuite, les résultats expérimentaux étant décevants, notamment à cause de certaines contraintes liées à l'algorithme (pas de mémorisation, recherche d'un individu unique, croisement), nous avons proposé un autre algorithme.

Cet algorithme original, inspiré du phénomène d'adaptation bactériologique, et appelé algorithme bactériologique, permet de résoudre les problèmes rencontrés avec l'algorithme génétique lors de l'optimisation de cas de test. Les apports importants de cet algorithme, sont la recherche d'un ensemble de solutions plutôt qu'une solution unique et la suppression de l'opération de croisement pour ne garder que la mutation comme opérateur pour la génération de nouvelles informations. Du fait de la limitation du nombre de paramètres, cet algorithme est plus facile à réutiliser et à généraliser. Pour l'optimisation d'un ensemble de cas de test, il a permis d'améliorer le score de mutation d'un ensemble initial rapidement, en générant un nombre raisonnable de cas de test supplémentaires.

De nouveaux travaux sont en cours pour la validation de cet algorithme sur de nouvelles études de cas et avec d'autres fonctions d'utilité (*e.g.*, couverture de code). Ces études devraient permettre aussi de généraliser le processus de génération bactériologique, et d'affiner les différents paramètres du modèle.

4
Robustesse et diagnosabilité : impact de la conception par contrat sur un assemblage de composants

La conception par contrat permet de définir précisément les obligations et devoirs des différents composants dans un système logiciel. Cette technique permet notamment d'embarquer la spécification de chaque opération dans le programme et de vérifier la cohérence de l'état interne du système au cours de l'exécution. De nombreux travaux reconnaissent l'importance d'une telle approche pour la conception de systèmes fiables, et pour la détection d'erreurs [Rosenblum"95; Carrillo-Castellon"96; Voas"99; Findler"01; Nordby"02], mais l'estimation de l'apport qualitatif des contrats pour un programme est rarement étudiée.

Dans le cadre des travaux présentés dans cette thèse, nous étudions la contribution des contrats en tant qu'oracle pour le test et leur impact sur la robustesse et la diagnosabilité. Tout d'abord, nous analysons la capacité des contrats à détecter des erreurs dans un composant à travers une mesure de robustesse. Nous proposons aussi un modèle qui prend en compte le fait que, dans le cadre d'un assemblage de composants, les contrats embarqués dans un composant peuvent détecter des erreurs présentes dans un autre composant fournisseur de services. Ensuite, nous étudions l'impact des contrats pour le diagnostic, *i.e.*, l'amélioration de la localisation d'une faute en présence de contrats. Pour cela, nous proposons un modèle de la mesure de diagnosabilité en présence de contrats.

Les travaux présentés dans ce chapitre concernent de manière équivalente tant le test de programme que la mesure de logiciels. Une grande partie des travaux présentés ici est donc dédiée à l'élaboration précise et non ambiguë d'une mesure. Pour cela, nous proposons une méthodologie générique pour l'élaboration d'une mesure, puis nous définissons les mesures de robustesse et diagnosabilité dans ce cadre. L'approche

consiste essentiellement à identifier les attributs participant à la définition de la mesure, puis à définir un modèle en fonction de ces attributs. Enfin, des expériences permettent de paramétrer ces attributs et d'observer ensuite le comportement de la mesure pour différentes valeurs des attributs obtenues.

4.1 Conception par contrat et élaboration d'une mesure

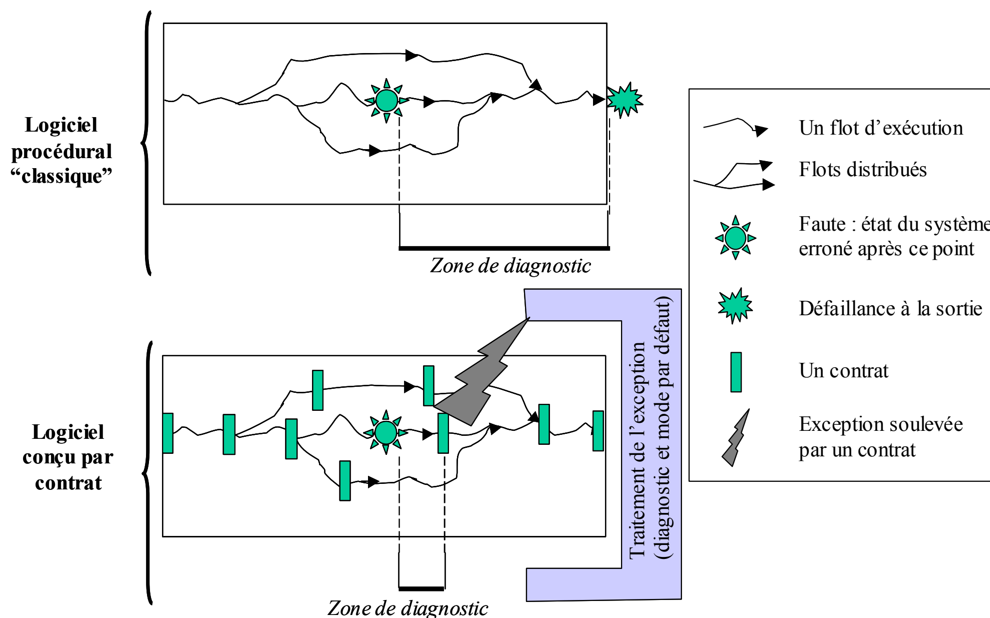


Figure 35 – Les contrats pour la diagnosabilité

4.1.1 Les contrats pour la robustesse et la diagnosabilité

Nous voulons maintenant mesurer l'apport d'une conception par contrat et déterminer leur influence sur la qualité finale du logiciel de deux manières :

– puisqu'une exception est soulevée lorsqu'un contrat est violé, ceux-ci se comportent comme des assertions classiques : un état erroné au cours de l'exécution peut être détecté automatiquement, et la défaillance qui se serait produite peut être évitée. De manière intuitive, il apparaît que les contrats participent à la robustesse du logiciel. La question est alors de savoir dans quelles proportions les contrats participent à la robustesse en fonction de leur « force » et de leur nombre

– lorsqu'un contrat est violé, il indique l'endroit dans le code où un état erroné a été détecté. Si aucun contrat n'est présent dans le logiciel, la défaillance est détectée plus tard, et se propage jusqu'à la sortie du système. Puisque la zone pour le diagnostic (le nombre d'instructions parmi lesquelles il faut chercher l'erreur) est réduite en présence de contrats, il apparaît que les contrats simplifient la tâche de diagnostic. La question est alors de savoir dans quelles

proportions les contrats participent à la diminution de l'effort de diagnostic en fonction de leur « force » et de leur nombre

Comme le montre la Figure 35, un système conçu par contrats devrait permettre une détection précoce d'une faute (pendant qu'elle se propage vers la sortie du logiciel). Ceci aiderait à la localisation de la faute en réduisant la zone pour le diagnostic. Ce qui rend les contrats intéressants, c'est que leur efficacité ne dépend pas d'une éventuelle répartition ou distribution de l'exécution du logiciel. Même si les fautes sont difficiles à localiser dans un environnement réparti, les contrats ont de grandes chances de détecter une erreur à proximité de l'instruction erronée.

4.1.2 Elaboration d'une mesure

La littérature insiste sur les difficultés d'établir une mesure valide [Fenton"86; Shepperd"93; Kitchenham"95; Briand"96]. Puisque les mesures que nous étudions ici apparaissent comme assez abstraites (au sens où il n'y a pas d'attribut mesurable immédiat qui la détermine), nous proposons une axiomatisation de ces mesures [Shepperd"93]. La Figure 36 illustre le processus pour l'élaboration d'une mesure. Tout d'abord, le facteur qui va être mesuré est décrit de manière informelle, et les attributs mesurables qui l'influencent sont identifiés. Puis les propriétés intuitives sur le comportement du facteur doivent être exprimées en fonction des attributs choisis : c'est ce que nous appelons les axiomes. Un axiome est une propriété attendue de la mesure qui apparaîtra dans le modèle mathématique. C'est ce qui fait le lien entre l'approche intuitive et l'approche formelle nécessaire à la validation théorique. Un modèle formel de la mesure peut alors être proposé. Ce modèle complète les axiomes. Le rôle de l'axiomatisation est double : elle offre un cadre commun pour l'évaluation de plusieurs mesures et elle permet de passer de l'idée intuitive à l'élaboration de la mesure. Un aspect secondaire non négligeable consiste à faire entrer dans le champ de la discussion la manière de mesurer. Grâce à l'explicitation des propriétés intuitives et à leur correspondance dans un modèle mathématique, les limites de la mesure apparaissent clairement.

Puisque les axiomes formalisent l'essentiel des propriétés attendues d'une mesure, ils définissent quels systèmes peuvent être comparés, et les caractéristiques génériques que ces mesures doivent vérifier. Les axiomes constituent la base de l'évaluation théorique que nous avons conduite pour vérifier l'adéquation de la mesure proposée avec un logiciel réel. L'évaluation théorique précède l'évaluation empirique puisqu'elle est moins coûteuse en temps et qu'elle permet de montrer la cohérence interne du modèle.

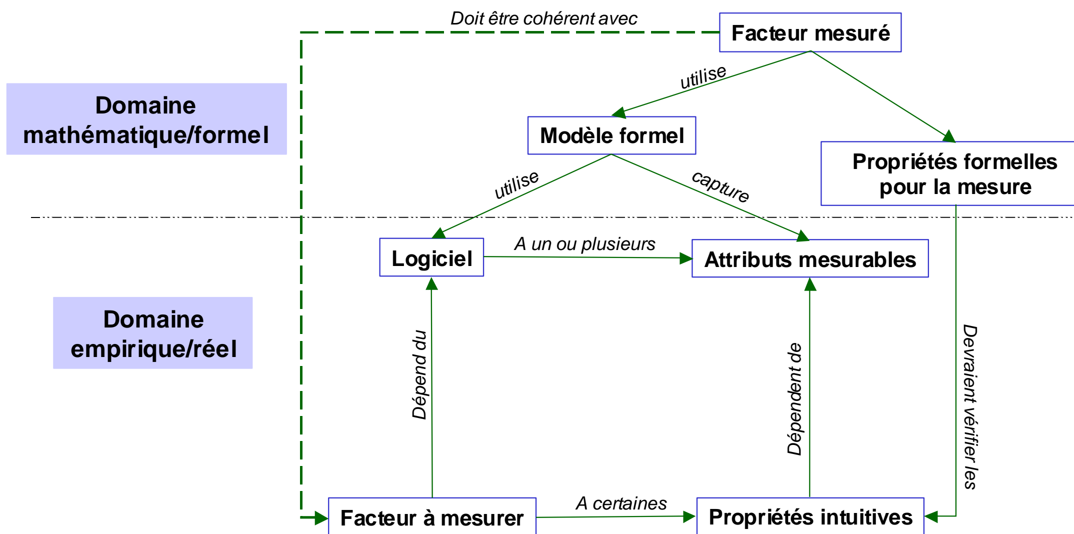


Figure 36 – Élaboration d'une mesure

4.2 Mesure de la robustesse

Nous présentons ici, un modèle qui lie la robustesse d'un composant logiciel et ses contrats. Une mesure de l'efficacité des contrats est proposée et l'amélioration de la robustesse grâce aux contrats est expliquée.

4.2.1 Définitions

Avant de définir la mesure de robustesse, nous revenons sur les notions de faute, erreur et défaillance qui sont utilisées par la suite. Comme nous l'avons dit à la section 2.1.1, une *faute* désigne la cause d'une erreur, une *erreur* est la partie de l'état d'un système qui peut entraîner une défaillance, et une *défaillance* est un événement survenant lorsque le service délivré dévie de l'accomplissement de la fonction du système. Dans le contexte particulier de l'étude présentée ici, une faute est une instruction ou un ensemble d'instructions (ou même l'absence d'instruction) qui peut entraîner une erreur. Une erreur est l'état d'un objet (l'ensemble des valeurs des attributs de l'objet) dans le système qui peut déclencher une défaillance. Par exemple, observons une version erronée de la classe BANKACCOUNT donnée Figure 37. L'instruction `balance = balance – sum` de la méthode `deposit`, devrait être `balance = balance + sum`. Cette instruction correspond à une faute si elle est exécutée, l'attribut `balance` de la classe BANKACCOUNT se voit affecter une mauvaise valeur. L'état de l'objet après l'exécution de cette méthode est une erreur. Cette erreur peut effectivement entraîner une défaillance puisque la valeur de `balance` n'est pas cohérente avec la valeur attendue spécifiée par la postcondition.

```
class BankAccount{
            float balance;
            public deposit (float sum){balance=balance-sum;}
            …
}

context BankAccount::deposit
  pre self.amount>0
  post self.balance = self.balance@pre + amount
```

Figure 37 – Exemple pour une faute, une erreur et une défaillance

Il est maintenant possible de définir la robustesse, tout d'abord de manière intuitive, puis de façon plus précise en distinguant différent niveaux de granularité pour la mesure de robustesse dans le contexte particulier de la conception par contrat.

Robustesse (définition informelle) : *La robustesse d'un composant C_i est définie comme la capacité de ce composant à fonctionner même dans des conditions anormales. Cette capacité permet d'éviter les défaillances catastrophiques. La robustesse peut aussi être vue comme la probabilité qu'une erreur soit détectée et traitée par un mécanisme d'exception sachant qu'une défaillance se produit certainement sinon.*

Nous considérons que l'attribut principal pour la robustesse est l'aptitude d'un logiciel à détecter des erreurs internes.

Robustesse isolée (RobIsolée(C_i)): *la robustesse isolée RobIsolée(C_i) d'un composant C_i dans un système S est définie comme la probabilité qu'une erreur soit détectée par C_i sachant que cette erreur aurait provoqué une défaillance. Inversement, la « faiblesse » d'un composant est égale à la probabilité que l'erreur ne soit pas détectée.*

Cette définition introduit la notion d'*erreur interne*. Une erreur interne dans un composant C_i, correspond à une valeur erronée d'un ou plusieurs attributs du composant C_i. Nous nous intéressons aux contrats et aux assertions pour la détection d'erreurs. Une erreur est donc considérée comme détectée si un contrat est violé à cause de cette erreur. Par exemple, la faute de la Figure 37 entraîne une erreur interne dans la classe BANKACCOUNT qui est détectée par la postcondition de la méthode `deposit`. De nombreux composants ne peuvent pas être exécutés directement (classes abstraites ou génériques, frameworks). Néanmoins, des contrats peuvent être définis pour ces composants, permettant ainsi la détection d'erreurs internes de leurs instances ou de leur client.

Robustesse globale (Rob) : *la robustesse globaleRob(S) d'un système S composé d'un ensemble de composants interconnectés est la probabilité qu'une erreur interne soit détectée par un de ces composants.*

Autrement dit, la robustesse globale correspond à la probabilité qu'un contrat détecte l'erreur lorsque l'état du programme devient erroné. Une faute dans un composant utilisé par un système peut être détectée soit par le composant lui-même soit par un de ces clients ou une spécialisation de ce composant. Il apparaît alors que la robustesse globale ne peut pas être déduite directement de la robustesse locale. Notre hypothèse est qu'il existe une relation entre la robustesse locale et la robustesse globale mais que des informations supplémentaires sur l'architecture sont nécessaires pour obtenir la valeur de la robustesse globale. En nous appuyant sur un diagramme de classes UML pour décrire l'architecture d'un logiciel, nous repérons les attributs pertinents qui peuvent être extraits de l'architecture. Ces attributs permettent de calculer la robustesse globale fondée sur la robustesse locale. Nous avons maintenant besoin de définir la robustesse locale d'un composant utilisé dans un système.

Robustesse locale (RobLoc) : *la robustesse locale RobLoc(C_i,S) d'un composant C_i dans un système S est définie comme la probabilité qu'une erreur soit détectée par C_i ou un autre composant dans le système S, sachant que cette erreur aurait provoqué une défaillance.*

Il est nécessaire de faire la distinction entre la robustesse isolée et la robustesse locale. En effet, si une faute dans un composant C_i n'est pas détectée par ses propres contrats, elle peut se propager à travers le système. L'erreur peut alors être détectée par les contrats d'autres composants. La robustesse d'un composant dans un assemblage de composants est donc différente de la robustesse du composant en-dehors de tout contexte. La robustesse locale et la robustesse isolée sont des mesures locales à un composant, alors que la robustesse globale concerne tout le système. L'axiomatisation est ainsi décomposée en axiomes locaux et globaux.

4.2.2 Axiomatisation

Les axiomes formalisent les propriétés attendues d'une mesure. A partir des définitions énoncées dans la section précédente, nous proposons trois ensembles d'axiomes : les axiomes globaux, les axiomes locaux (pour la robustesse locale et isolée) et les axiomes qui lient les mesures globales aux mesures locales. Nous nous appuierons ensuite sur ces axiomes pour la vérification théorique de la cohérence entre les mesures et les propriétés attendues.

a Axiomes locaux de robustesse

ALR1 – Comparaison entre composant. Tous les composants d'un système sont comparables en termes de robustesse locale ou isolée.

ALR2 – Composant sans contrat (ni assertion). Les composants qui n'ont pas de contrats (ni d'assertions ni aucun autre mécanisme de détection d'erreur) ont une robustesse locale nulle.

Les axiomes suivants définissent le comportement attendu, de manière intuitive, lors de certaines opérations sur le modèle conceptuel : assemblage de systèmes, ajout de contrats et amélioration des contrats.

- *Assemblage* : regroupe toute opération qui consiste à connecter deux systèmes pour en créer un nouveau (par exemple en utilisant l'héritage, ou des relations d'utilisation)
- *Ajout de contrats* : l'opération qui consiste à définir des contrats pour un composant dans un système
- *Amélioration des contrats* : opération qui consiste à ajouter une nouvelle clause dans un contrat existant, pour vérifier une propriété qui n'était pas encore vérifiée. Un contrat est donc amélioré uniquement s'il vérifie plus de propriétés pour un composant.

Par exemple, une précondition pour la méthode `withdraw` de la classe BANKACCOUNT peut être:

```
context BankAccount::withdraw(amount:float):void
pre amount>0
```

Une amélioration de cette précondition peut consister en la précondition suivante

```
context BankAccount::withdraw(amount:float):void
pre amount>0 and amount ≤ balance-overdraft
```

ALR3 – Assemblage de systèmes. La robustesse isolée d'un composant inclus dans un système S1 n'est pas modifiée par l'assemblage avec un système S2. La robustesse locale du composant ne peut pas diminuer.

ALR4 – Ajout de contrat. La robustesse locale et la robustesse isolée d'un composant ne peuvent pas décroître lors de l'ajout de contrat dans le système.

ALR5 – Amélioration des contrats. L'amélioration d'un contrat dans un composant C_i doit améliorer sa robustesse locale et sa robustesse isolée. La robustesse locale et la robustesse isolée des autres composants ne peut pas diminuer.

b Axiomes globaux de robustesse

AGR1 – Comparaison entre systèmes. Deux systèmes sont toujours comparables en termes de robustesse.

AGR2 – Assemblage de systèmes. La robustesse globale d'un système obtenu par l'assemblage de deux systèmes S1 et S2 ne peut pas être plus faible que la plus faible des robustesses de S1 et S2. Ainsi, la robustesse d'un système S_3, composé par l'assemblage des systèmes doit vérifier : $Rob(S_3) \geq min(Rob(S_1), Rob(S_2))$.

AGR3 – Ajout de contrat. Pour un système, sa robustesse globale ne peut pas décroître lors de l'ajout d'un contrat.

4.2.3 Hypothèses et modèle mathématique

Selon notre définition, la robustesse d'un composant en-dehors de tout système correspond à l'efficacité de ses contrats, et nous l'avons nommée la robustesse isolée. La robustesse d'un composant utilisé dans un système est améliorée par les contrats de ses clients. Nous utilisons la notion de dépendance de test, introduite dans [Le Traon"00a], pour expliquer la relation entre un composant et ses clients pour la robustesse.

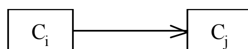


Figure 38 – Dépendance de test

Dépendance de test. *Un composant C_i dépend de C_j pour le test s'il utilise des services fournis par C_j. Cette relation de dépendance est notée : $C_iR_{TD}C_j$. Nous avons besoin d'exprimer cette notion de clientèle (qui s'étend à l'héritage) pour établir le modèle de la mesure de robustesse globale en fonction des robustesses isolées de l'architecture.*
Par exemple sur la Figure 38, C_i utilise les services de C_j (Figure 38), on note alors $C_iR_{TD}C_j$.

Probabilité Det(C_i, C_j). *Si $C_iR_{TD}C_j$, alors la probabilité que les contrats de C_i détectent une erreur lors d'une utilisation par C_j est notée $Det(C_i, C_j)$. Le cas limite pour cette mesure est $Det(C_i, C_i)$ qui correspond à la robustesse isolée du composant C_i. Par ailleurs, si C_i et C_j sont dans des systèmes différents (S_1 et S_2), on notera cette probabilité $Det((C_i, S_1), (C_j, S_2))$.*

Remarque : une probabilité $Det(C_i, C_j)$ non nulle signifie que la robustesse d'un composant embarqué dans un système sera améliorée grâce au couplage entre les composants. Il est alors intéressant de noter que dans le cas d'un système conçu avec des contrats, un couplage important va permettre d'améliorer la qualité globale du système alors que c'est habituellement considéré comme un signe d'une mauvaise conception.

Dans la section suivante, nous proposons une méthode pour évaluer la robustesse d'un composant utilisé dans un système, ainsi que pour calculer la probabilité $Det(C_i, C_j)$.

Même si la relation de dépendance de test est transitive, nous ne considérons que les fautes détectées par un composant directement dépendant du composant erroné. L'héritage est un cas spécial de dépendance de test dont l'impact sur la robustesse ne nous apparaît pas prévisible intuitivement. Aussi, au cours des expérimentations, avons-nous considéré deux hypothèses : une première approche pessimiste qui considère que la probabilité $Det(C_i, C_j)$ est nulle dans le cas de l'héritage, et une seconde pour laquelle nous considérons que la probabilité est la même dans le cas de l'héritage que dans le cas d'une relation de clientèle.

La robustesse RobLoc(C, S) (1- FaibLoc(C, S)) d'un composant C dans un système S est la probabilité qu'une faute dans C soit détectée par les contrats de C ou par une utilisation de C par d'autres composants. Pour calculer cette probabilité, nous calculons FaibLoc(C, S). La probabilité FaibLoc(C, S) correspond à la probabilité qu'une faute due à C ne soit pas détectée par C multipliée par la probabilité que l'erreur ne soit pas détectée par les clients de C.

Robustesse locale. *La robustesse locale est calculée de la manière suivante :* $RobLoc(C_i, S)$ *(1-* $FaibLoc(C_i,S)$*) où* $FaibLoc(C_i,S) = \prod_k (1 - Det(C_k, C_i))$*, avec* $k \,/\, C_k\, R_{TD}\, C_i$.

Robustesse globale. *La robustesse globale Rob d'un système s'exprime à partir de la robustesse locale de la manière suivante :*

$$Rob(S) = 1 - Faib(S) = 1 - \sum_{i=1}^{|S|} ProbErr(C_i, S) \cdot FaibLoc(C_i, S)$$

Pour le calcul de Rob(S), $ProbErr(C_i,S)$ est la probabilité qu'une défaillance provienne de C_i sachant qu'une défaillance apparaîtra forcément. La complexité du composant est une approximation de cette probabilité. Par ailleurs, on note |S| le nombre de composants dans le système S.

4.2.4 Démonstration des axiomes

La démonstration des axiomes locaux étant assez immédiate, nous détaillons ici la démonstration des deux axiomes globaux principaux : AGR2 et AGR3. Nous rappelons d'abord, les notations pour les mesures et définissons les profils :

- Rob(S) la robustesse globale d'un système S.

  Rob(S) : Système → [0…1]

- RobLoc(C, S) la robustesse locale d'un composant C dans un système S.

RobLoc(C, S) : Composant $\times$ Système $\rightarrow$ [0…1]

- FaibLoc(C, S) l'opposée de la robustesse locale de C dans S.

  FaibLoc(C, S) : Composant $\times$ Système $\rightarrow$ [0…1]

- ProbErr(C, S) la probabilité qu'une erreur soit localisée dans le composant C d'un système S. Nous faisons l'hypothèse que cette probabilité est uniforme pour un système : ProbErr(C, S) = 1/|S|.

- Det(C_i, C_j) la probabilité qu'un composant C_i détecte une erreur dans C_j.

  Det(C_i, C_j) : Composant $\times$ Composant $\rightarrow$ [0…1]

a Preuve de l'axiome AGR2 : assemblage de deux systèmes

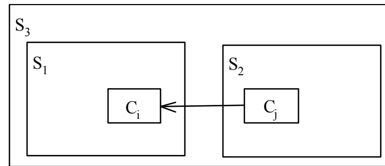


Figure 39 – Assemblage de deux systèmes

Nous voulons montrer ici, que la robustesse d'un système S_3, composé par l'assemblage des systèmes S_1 et S_2, n'est pas inférieure à la plus petite des robustesses de S_1 et S_2. En d'autres termes, nous voulons vérifier que : $Rob(S_3) \geq \min(Rob(S_1), Rob(S_2))$.

L'assemblage de deux systèmes S_1 et S_2 se fait par l'ajout d'une relation entre un nœud de S_1 et un nœud de S_2 comme l'illustre la Figure 39. La formule générale qui exprime la robustesse de S_3 est :

$$Rob(S_3) = 1 - \sum_{k=1}^{|S_3|} (ProbErr(C_k, S_3) \times FaibLoc(C_k, S_3))$$

Comme il n'y a pas d'ajout de nouveau nœud lors de l'assemblage de deux systèmes, on a : $|S_3| = |S_1| + |S_2|$, et donc :

$$Rob(S_3) = 1 - \sum_{i=1}^{|S_1|} (ProbErr(C_i, S_3) \times FaibLoc(C_i, S_3)) - \sum_{j=1}^{|S_2|} (ProbErr(C_j, S_3) \times FaibLoc(C_j, S_3))$$

Posons A et B tels que :

$$A = \sum_{i=1}^{|S_1|} (ProbErr(C_i, S_3) \times FaibLoc(C_i, S_3)), \text{ et } B = \sum_{j=1}^{|S_2|} (ProbErr(C_j, S_3) \times FaibLoc(C_j, S_3))$$

L'ajout de l'arc (C_i, C_j) a un impact sur la robustesse locale de C_i, mais ne modifie rien pour C_j. L'ajout de cet arc n'a donc pas d'effet sur le système S_2, auquel appartient C_j, on simplifie donc d'abord le terme B qui concerne S_2.

$$B = \sum_{j=1}^{|S_2|}\left(\frac{1}{|S_3|} \times FaibLoc(C_j, S_3)\right) = \frac{|S_2|}{|S_3|} \times \sum_{j=1}^{|S_2|}\left(\frac{1}{|S_2|} \times FaibLoc(C_j, S_3)\right)$$

Or, $FaibLoc(C,S_2) = FaibLoc(C,S_3)$ pour tout composant C dans S_2, donc :

$$B = \frac{|S_2|}{|S_3|} \times \sum_{j=1}^{|S_2|}\left(\frac{1}{|S_2|} \times FaibLoc(C_j, S_2)\right) \text{ et donc } B = \frac{|S_2|}{|S_3|} \times (1 - Rob(S_2)) .$$

On s'intéresse maintenant au terme A. Tout d'abord, de la même façon que pour le terme B, on obtient : $A = \frac{|S_1|}{|S_3|} \times \sum_{i=1}^{|S_1|}\left(\frac{1}{|S_1|} \times FaibLoc(C_i, S_3)\right)$.

Maintenant, si on désigne C_1 par le composant sur lequel la relation est rajoutée, on a alors : $\forall$ i $\in$ [2…| S_1|], $FaibLoc(C_i,S_1) = FaibLoc(C_i,S_3)$ et donc :

$$A = \frac{|S_1|}{|S_3|} \times \left(\sum_{i=2}^{|S_1|}\left(\frac{1}{|S_1|} \times FaibLoc(C_i, S_1)\right) + \frac{1}{|S_1|} \times \underbrace{FaibLoc(C_1, S_3)}_{\alpha}\right) .$$

$$\alpha = \prod_k (1 - Det((C_k, S_1), (C_1, S_1))) \times (1 - Det((C_1, S_2), (C_1, S_1)))$$

$$\Leftrightarrow \alpha = FaibLoc(C_1, S_1) \times (1 - Det((C_1, S_2), (C_1, S_1)))$$

Comme $1-Det((C_1,S_2), (C_1,S_1)) \leq 1$, alors :

$FaibLoc(C_1,S_3) \leq FaibLoc(C_1,S_1)$

$$\Rightarrow \frac{1}{|S_1|} \times FaibLoc(C_1, S_3) \leq \frac{1}{|S_1|} \times FaibLoc(C_1, S_1)$$

D'où : $A \leq \frac{|S_1|}{|S_3|} \times \sum_{i=1}^{|S_1|}\left(\frac{1}{|S_1|} \times FaibLoc(C_i, S_1)\right)$ et donc $A \leq \frac{|S_1|}{|S_3|} \times (1 - Rob(S_1))$

Maintenant que nous avons simplifié les termes A et B, revenons à l'expression de la robustesse globale de l'assemblage S_3.

$Rob(S_3) = 1-A-B$,

$$Rob(S_3) \geq 1 - \frac{|S_1|}{|S_3|} \times (1 - Rob(S_1)) - \frac{|S_2|}{|S_3|} \times (1 - Rob(S_2))$$

Donc pour montrer que $Rob(S_3) \geq \min(Rob(S_1), Rob(S_2))$, il faut montrer :

$$1 - \frac{|S_1|}{|S_3|} \times (1 - Rob(S_1)) - \frac{|S_2|}{|S_3|} \times (1 - Rob(S_2)) \geq \min(Rob(S_1), Rob(S_2))$$

Comme $|S_3| = |S_1|+|S_2|$, l'expression ci-dessus peut se réécrire de la manière suivante:

$$1-\frac{|S_1|}{|S_1|+|S_2|}\times(1-\mathrm{Rob}(S_1))-\frac{|S_2|}{|S_1|+|S_2|}\times(1-\mathrm{Rob}(S_2))\geq \min(\mathrm{Rob}(S_1),\mathrm{Rob}(S_2))$$

Considérons maintenant min(Rob(S_1), Rob(S_2)) = Rob(S_1) on veut maintenant montrer :

$$1-\frac{|S_1|}{|S_1|+|S_2|}\times(1-\mathrm{Rob}(S_1))-\frac{|S_2|}{|S_1|+|S_2|}\times(1-\mathrm{Rob}(S_2))\geq \mathrm{Rob}(S_1)$$

$$\frac{1}{|S_1|+|S_2|}\times(|S_1|+|S_2|-|S_1|+|S_1|\times\mathrm{Rob}(S_1)-|S_2|+|S_2|\times\mathrm{Rob}(S_2)-|S_1|\times\mathrm{Rob}(S_1)-|S_2|\times\mathrm{Rob}(S_1))\geq 0$$

$$\frac{1}{|S_1|+|S_2|}\times(|S_2|\times\mathrm{Rob}(S_2)-|S_2|\times\mathrm{Rob}(S_1))\geq 0$$

$$\frac{1}{|S_1|+|S_2|}\times(|S_2|\times(\mathrm{Rob}(S_2)-\mathrm{Rob}(S_1)))\geq 0$$

Comme min(Rob(S_1), Rob(S_2)) = Rob(S_1), alors Rob(S_2) - Rob(S_1) ≥ 0 et on a donc bien :

$$\frac{1}{|S_1|+|S_2|}\times(|S_2|\times(\mathrm{Rob}(S_2)-\mathrm{Rob}(S_1)))\geq 0$$

Nous avons ainsi montré Rob(S_3) ≥ min(Rob(S_1), Rob(S_2)).

b Preuve de AGR3 : Ajout de contrat

Nous voulons démontrer que l'ajout d'un contrat dans un système S ne fait pas décroître sa robustesse globale. Si on note S' le système après l'ajout d'un contrat, on veut montrer :

Rob(S') ≥ Rob(S).

L'ajout d'un contrat dans un composant C_i modifie ni sa robustesse isolée, ni la robustesse locale de ses composants serveurs. Par exemple sur la Figure 40, l'ajout d'un contrat dans C_1 modifie la robustesse isolée de C_1 ainsi que RobLoc(C_2), RobLoc(C_3), et la robustesse locale de tous ses serveurs.

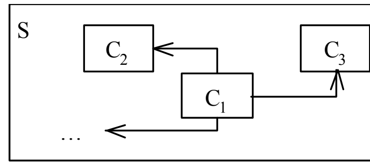


Figure 40 – Ajout d'un contrat dans C_1

Considérons l'ajout d'un contrat dans C_i, nous notons C'_i le composant obtenu après ajout du contrat. La robustesse isolée de C'_i est supérieure ou égale à celle de C_i :

$$\mathrm{Det}(C'_i, C'_i) \geq \mathrm{Det}(C_i, C_i) \qquad (1)$$

Par ailleurs, la probabilité que C'_i détecte une erreur dans un de ces serveurs est supérieure ou égale à celle que C_i détecte l'erreur : $\forall$ k / C_i R_{TD} C_k, on a $Det(C'_i, C_k) \geq Det(C_i, C_k)$, donc :

$$\prod_k (1 - Det(C'_i, C_k) \leq \prod_k (1 - Det(C_i, C_k)$$

i.e. :$\forall$ k / C_i R_{TD} C_k, on a $FaibLoc(C_k, S') \leq FaibLoc(C_k, S)$ (2)

Si on désigne par C_1 le composant modifié, et qu'on désigne ces serveurs par $C_2 \ldots C_m$, avec $m < |S'|$, alors Rob(S') s'écrit de la manière suivante :

$$Rob(S') = 1 - \sum_{i=m+1}^{|S'|} (ProbErr(C'_i, S') \times FaibLoc(C'_i, S')) + \sum_{i=1}^{m} (ProbErr(C'_i, S') \times FaibLoc(C'_i, S'))$$

Or, pour i > m, $FaibLoc(C'_i, S') = FaibLoc(C_i, S)$ car l'ajout de contrat n'a aucune influence sur la robustesse des composants d'indice i > m qui ne sont pas connectés au composant modifié. De plus, comme le nombre de composants dans S' ne change pas, on a $|S'| = |S|$ et donc $ProbErr(C'_i, S') = ProbErr(C_i, S)$. La robustesse Rob(S') se réécrit :

$$Rob(S') = 1 - \sum_{i=m+1}^{|S|} (ProbErr(C_i, S) \times FaibLoc(C_i, S)) + \sum_{i=1}^{m} (ProbErr(C'_i, S') \times FaibLoc(C'_i, S'))$$

Comme $C_1 R_{TD} C_2 \ldots C_1 R_{TD} C_m$, l'inégalité (2) nous dit que :

$\forall k \in [2 \ldots m]$, $FaibLoc(C_k, S') \leq FaibLoc(C_k, S)$.

De plus (1) nous dit que :

$Det(C'_1, C'_1) \geq Det(C_1, C_1)$, i.e. $FaibLoc(C_1, S') \leq FaibLoc(C_1, S)$.

On a alors :

$$\sum_{i=1}^{m} (ProbErr(C'_i, S') \times FaibLoc(C'_i, S')) \leq \sum_{i=1}^{m} (ProbErr(C_i, S) \times FaibLoc(C_i, S)),$$

et donc :

$$\sum_{i=m+1}^{|S|} (ProbErr(C_i, S) \times FaibLoc(C_i, S)) + \sum_{i=1}^{m} (ProbErr(C'_i, S') \times FaibLoc(C'_i, S'))$$
$$\leq \sum_{i=m+1}^{|S|} (ProbErr(C_i, S) \times FaibLoc(C_i, S)) + \sum_{i=1}^{m} (ProbErr(C_i, S) \times FaibLoc(C_i, S))$$

Finalement :

$$Rob(S') \geq 1 - \sum_{i=m+1}^{|S|} (ProbErr(C_i, S) \times FaibLoc(C_i, S)) + \sum_{i=1}^{m} (ProbErr(C_i, S) \times FaibLoc(C_i, S))$$

c'est-à-dire : Rob(S') ≥ Rob(S).

4.2.5 Expériences pour paramétrer le modèle

Différentes expériences sur un système orienté objets, nous ont permis de fixer des valeurs pour les différents paramètres du modèle. Nous commençons par mesurer la robustesse isolée des différentes classes du système en utilisant l'analyse de mutation. Puis, toujours à base de mutation, nous évaluons la probabilité $Det(C_i, C_j)$ (probabilité de détection d'erreurs d'un composant j par un de ses clients i).

a Estimation de l'efficacité des contrats (robustesse isolée)

```
+BankAccount:: transfer(real sum, BankAccount account){
   balance=balance-sum;
   account.deposit(sum);
}

context BankAccount::transfer(real sum,BankAccount account)
pre sum>0 and balance-sum>=overdraft
post balance = balance@pre - sum
```

```
Mutant1 (NOR)
public void transfer(float sum, TP2Bank.BankAccount account){
   account.deposit(sum);
}
Mutant2 (NOR)
public void transfer(float sum, TP2Bank.BankAccount account){
   balance=balance-sum;
}
Mutant3 (AOR)
public void transfer(float sum, TP2Bank.BankAccount account){
   balance=balance+sum;
   account.deposit(sum);
}
Mutant4 (VCP)
public void transfer(float sum, TP2Bank.BankAccount account){
   balance=balance-sum;
   account.deposit(sum-1);
}
Mutant5 (VCP)
public void transfer(float sum, TP2Bank.BankAccount account){
   balance=balance-sum-1;
   account.deposit(sum);
}
```

Figure 41 – Exemples pour le calcul de la robustesse isolée

Pour calculer la qualité des contrats des classes d'un système, c'est-à-dire la robustesse isolée d'une classe, nous avons utilisé l'analyse de mutation deux fois. Au cours de ces deux analyses, nous utilisons deux techniques différentes pour l'oracle : la différence de comportements et les contrats (cf. section 3.1.1). La première analyse permet de générer un ensemble de cas de test efficaces dont on sait qu'ils tuent 100% des

mutants. La seconde permet d'évaluer l'efficacité des contrats avec les cas de test efficaces. Remarquons que le score de mutation obtenu avec cette seconde analyse est bien une estimation de la qualité des contrats. En effet, puisque cette analyse utilise un ensemble de cas de test efficaces à 100%, un score de mutation plus faible en utilisant les contrats comme oracle nous assure qu'un mutant ne peut être vivant qu'à cause de contrats trop faibles. Donc, moins les contrats sont efficaces, plus il reste de mutants vivants. Le nouveau score de mutation, avec les contrats comme oracle, est donc un estimateur valide de l'efficacité des contrats.

Au cours de la première analyse, nous avons utilisé l'oracle par différence de comportement entre la classe initiale et le mutant. Cette analyse se fait de manière incrémentale. Un premier score de mutation est calculé sur un ensemble de cas de test initial. Si ce score n'est pas satisfaisant, des cas de tests supplémentaires sont générés pour l'améliorer. A la fin de cette première analyse, nous obtenons un ensemble cas de test avec un bon score de mutation pour une classe (>95%).

La seconde analyse est exécutée en n'utilisant que les contrats comme fonction d'oracle, et en utilisant l'ensemble de cas de test final obtenu lors de la première analyse. Le score de mutation obtenu lors de cette analyse correspond à la proportion de mutants que les contrats peuvent détecter. Ce score est utilisé comme une estimation de la robustesse isolée d'un composant. De la même façon qu'au cours de la première analyse, si cette robustesse initiale n'est pas satisfaisante, les contrats sont améliorés.

Par exemple, la Figure 41 montre la méthode `transfer` de la classe BANKACCOUNT, avec ces pré et post conditions, ainsi que cinq mutants pour la méthode (en tout, 12 mutants on été générés). Cette méthode transfère la somme `sum` du compte courant au compte `account`. La postcondition peut tuer les mutants 1 et 3 et l'invariant tue le mutant 5 avec l'appel de méthode suivant : transfer(balance-overdraft, bAcc1). Le score de mutation est de 60% dans ce cas particulier. Pour obtenir la robustesse isolée pour la classe BankAccount, il est nécessaire de générer les mutants pour toutes les méthodes de la classe et calculer le score global.

b Calcul de Det(C_i, C_j)

Une fois obtenue la robustesse isolée pour chaque classe, on peut calculer la probabilité Det(C_i, C_j). Cette valeur, pour un composant C_i, est mesurée en injectant des fautes dans les fournisseurs de C_i (les composants utilisés par C_i), puis en exécutant les cas de test de C_i en utilisant les fournisseurs mutés. La proportion de mutants tués correspond à Det(C_i, C_j). Par exemple, les cas de test de BANK peuvent être exécutés avec les mutants de BANKACCOUNT, et dans la suite nous illustrons le cas d'une erreur issue de la méthode `transfer()` du fournisseur BANKACCOUNT et détectée par les contrats de BANK. Considérons les mutants pour la classe BANKACCOUNT donnés sur la Figure 41 et la méthode `transferInterest()` de la classe BANK (Figure 42), et

considérons un cas de test qui appelle `transferInterest()` et pour lequel la méthode `transfer()` est appelée avec une valeur pour l'intérêt de `accountFrom.balance-accountFrom-overdraft`. Si on ignore l'invariant de BANKACCOUNT, mutant5 n'est pas tué par la postcondition de `transfer()` mais est tué par la postcondition de `transferInterest()`. Avec cette donnée de test particulière, l'attribut `balance` est supérieur ou égal à `overdraft-1` après le transfert du mutant5. La postcondition de la méthode `transfer()` ne vérifie pas que balance est supérieure ou égale à `overdraft`, donc l'erreur n'est pas détectée localement. Par contre, la postcondition de l'appelant `transferInterest()` de BANK vérifie `accountFrom.getBalance() ≥ accountFrom.getOverdraft()` et détecte ainsi l'erreur introduite dans mutant5.

```
+Bank:: transferInterest(BankAccount accountFrom, BankAccount accountTo){
   real interest=accountFrom.getInterest();
   accountFrom.tranfer(interest,accountTo);
}

context BankAccount::transferInterest(BankAccount accountFrom, BankAccount accountTo)
post accountFrom.getBalance()>=accountFrom.getOverdraft()
```

Figure 42 – Exemple pour le calcul de Det(C_i, C_j)

c Les paramètres du modèle

On peut maintenant définir les différents paramètres du modèle en terme d'analyse de mutation.

RobIsolée(C_i) correspond au score de mutation obtenu lors d'une analyse de mutation utilisant les contrats comme oracle

Det(C_i, C_j)= proportion de mutants de C_j détectés par C_i, C_j étant un serveur de C_i.

ProbErr(C_i, S) = 1/|S|, |S| étant le nombre de composants/classes dans le système.

Nous pouvons ainsi mesurer, de manière expérimentale, des valeurs pour ces différents paramètres, ce qui nous permet de calculer les valeurs de la robustesse locale et de la robustesse globale. Nous rappelons l'expression de RobLoc(C_i, S) (1-FaibLoc(C_i, S)), ainsi que la robustesse globale :

$$\mathrm{FaibLoc}(C_i, S) = \prod_k (1 - \mathrm{Det}(C_k, C_i)), k / C_k R_{TD} C_i$$

$$\mathrm{Rob}(S) = 1 - \mathrm{Faib}(S) = \sum_{i=1}^{|S|} \mathrm{ProbErr}(C_i, S) \times \mathrm{FaibLoc}(C_i, S)$$

La section suivante décrit une étude de cas qui permet de calibrer les valeurs des paramètres.

d Etude de cas

A partir d'un système pour lequel chaque classe a un ensemble de cas tests, les buts de l'étude sont les suivants :

1. améliorer l'efficacité des contrats en utilisant cette approche
2. évaluer la capacité d'un composant à détecter des erreurs dues à ces fournisseurs.

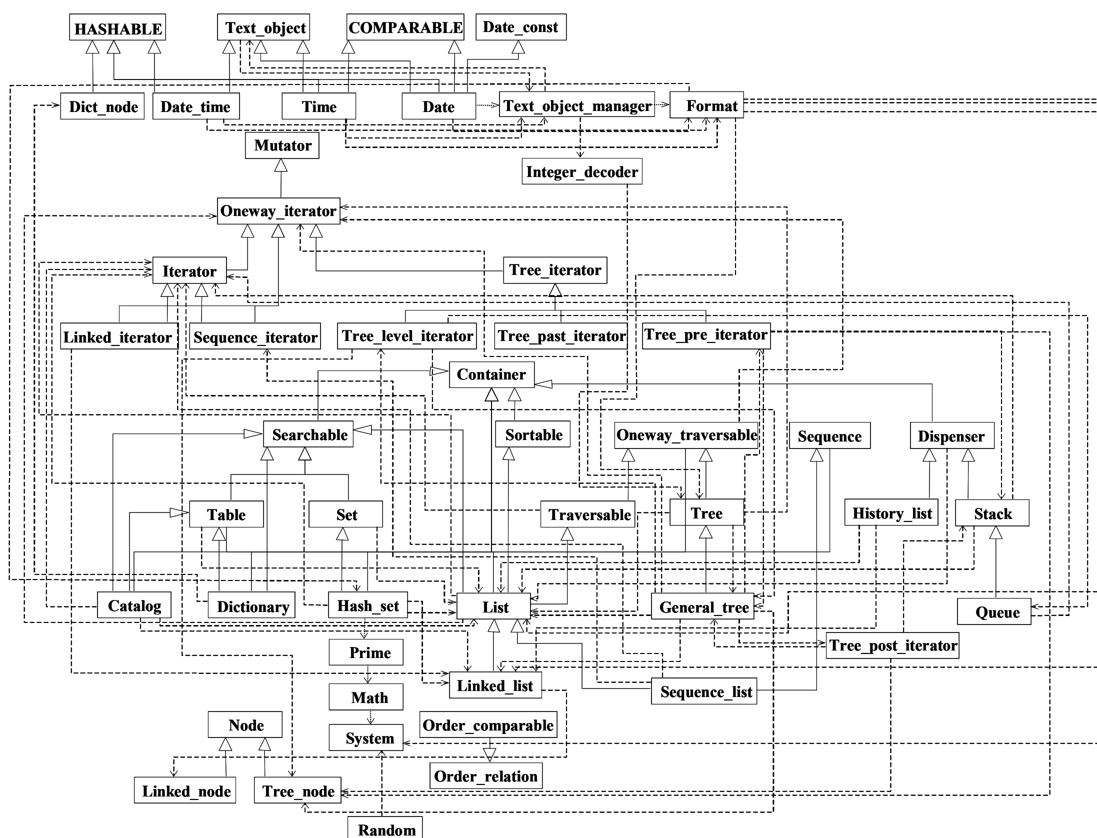


Figure 43 – Bibliothèque Pylon pour le langage Eiffel

Une étude de cas a été exécutée sur la bibliothèque Pylon pour le langage Eiffel. Cette bibliothèque implante plusieurs structures de données telles que la liste, le dictionnaire… La Figure 43 montre le diagramme de classes de cette librairie qui est composé de 50 classes et 134 relations entre ces classes. La complexité de cette librairie est suffisante pour illustrer notre approche et offrir des résultats intéressants. Nous avons utilisé l'outil µSlayer, dédié au langage Eiffel, pour l'analyse de mutation. Cet outil injecte des fautes dans une classe (ou un ensemble de classes) Eiffel sous test, exécute les tests sur chaque mutant et détermine lesquels sont tués par les tests. Cet outil permet d'avoir une approche incrémentale lors de la génération des cas de test (il permet de conduire une nouvelle analyse uniquement sur les mutants encore vivants) et peut être paramétré. L'utilisateur choisit le nombre de mutants qu'il veut générer, ainsi que le type d'erreurs qu'il veut injecter.

Le Tableau 4 donne la qualité des contrats initiaux et les scores obtenus après amélioration. Il apparaît que la robustesse isolée des classes a été améliorée de manière significative (la meilleure amélioration a permis de passer de 25% à 100%). A la fin du processus d'amélioration des contrats, un composant contractualisé a une meilleure capacité à détecter des erreurs (entre 75% et 100% dans notre cas). De plus, cette approche permet de détecter les endroits dans le composant pour lesquels les contrats sont trop faibles.

La qualité des contrats correspond à la capacité de ceux-ci à détecter des erreurs. La généralisation de cette approche serait de pouvoir écrire des contrats suffisamment efficaces pour les utiliser comme oracle pour le test (et éviter l'écriture d'oracles explicites pour chaque cas de test). Cependant, les résultats montrent la limite des contrats pour le test, puisqu'on n'a pas réussi à tuer tous les mutants avec les contrats. Les plupart des erreurs qui ne sont pas détectées par les contrats affectent l'état global du composant. Il est donc difficile, pour des contrats locaux à la fin ou au début d'une routine, de détecter ces erreurs. Par exemple, il est difficile d'écrire des contrats pour une méthode retrait d'une classe pile qui vérifie que l'élément retiré a bien été inséré préalablement. Les invariants de classe sont mieux adaptés pour la vérification de propriétés globales.

	Minimum	Maximum	Moyen
% mutants tués (contrats initiaux)	**17%**	**83%**	**58,5%**
% mutants tués après amélioration des contrats	**72%**	**100%**	**87,5%**

Tableau 4 – Résultats principaux pour l'amélioration des contrats

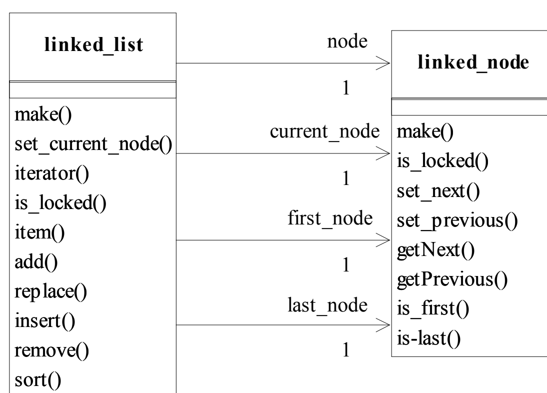


Figure 44 – Extraits de la bibliothèque Pylon

Pour mesurer les valeurs de Det(C_i, C_j), les mutants pour tous les serveurs d'une classe C_j sont générés, et les scores de mutation pour les cas de test de C_i sur l'ensemble

des mutants concernant les méthodes utilisées par C_i sont calculés. Sur l'exemple de la Figure 44, LINKED_LIST utilise LINKED_NODE et la classe LINKED_LIST appelle toutes les méthodes de LINKED_NODE. Le calcul de Det(LINKED_LIST, LINKED_NODE) consiste à générer tous les mutants pour LINKED_NODE et à exécuter les cas de test de LINKED_LIST. Le score de mutation obtenu correspond à la proportion de mutants de LINKED_NODE détectés par les contrats de LINKED_LIST. Ce score est la valeur Det(LINKED_LIST, LINKED_NODE).

	Min.	Max.	Moyen
% mutants du serveur tués par le client	**50%**	**84%**	**69%**

Tableau 5 – Mesures de Det(C_i, C_j)

Pour les mesures de Det(C_i, C_j), toutes les classes du système sont exécutées avec leur contrats optimisés (la valeur moyenne pour la robustesse isolée est de 87%), et les résultats sont présentés dans le Tableau 5.

4.2.6 Application de la mesure sur trois systèmes réels et résultats

Pour illustrer l'intérêt de la conception par contrats pour l'amélioration de la robustesse, nous l'avons appliquée a posteriori à trois études de cas.

- Un système de commutateur pour les télécommunications (SMDS : Switched MegaBits Data Service).
- La bibliothèque graphique InterViews composée de 146 classes et 420 relations [Kung"96]
- La bibliothèque Pylon

Pour ces trois systèmes, nous estimons l'évolution de la robustesse globale du système en fonction de l'évolution de la robustesse isolée des composants. Sur la Figure 45, nous avons considéré que la probabilité Det(C_i, C_j) est proportionnelle à la robustesse isolée du composant i (i.e. Det(C_i, C_j) = K*RobIsolée(C_i)). En utilisant les résultats du Tableau 4 et du Tableau 5, nous avons fixé le coefficient de proportionnalité à 0,8. De plus, pour l'évolution de la robustesse des systèmes sur la Figure 45, nous avons fait l'hypothèse que les dépendances dues à l'héritage n'ont pas d'effet sur la robustesse globale.

Les résultats montrent que s'il n'y a aucun contrat dans le système, la robustesse est nulle, mais que le simple ajout de contrats, même d'efficacité faible, améliore la robustesse globale rapidement. En effet, pour une valeur robustesse isolée comprise entre 0 et 0,2, la robustesse globale obtenue augmente très rapidement. Pour des contrats de qualité moyenne (robustesse isolée entre 0,2 et 0,6), la robustesse globale augmente proportionnellement à la robustesse isolée. Pour des bons contrats (robustesse isolée entre 0,6 et à 0,85) la robustesse globale augmente moins vite, mais tout de même de

manière significative (elle passe de 0,78 à 0,94). Enfin, on observe que pour les trois courbes, faire passer la robustesse isolée de 0,85 à 1, ce qui correspond aux transformations les plus coûteuses, n'est pas très intéressant en terme d'amélioration de la robustesse. Par exemple, la robustesse globale d'InterViews est déjà de 0,94 quand la robustesse isolée des composants est égale à 0,85.

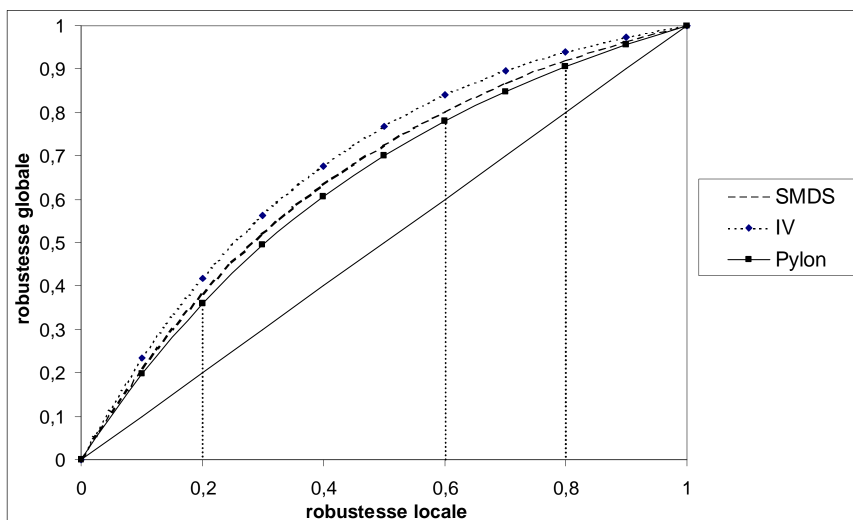


Figure 45 – Évolution de la robustesse globale en fonction de la robustesse locale des composants

Les différents résultats entre les systèmes correspondent à différentes densités dans les relations de dépendance. Comme on l'a remarqué plus tôt la robustesse locale d'un composant peut être améliorée par les contrats de ses clients, et l'amélioration de la robustesse locale participe à l'amélioration de la robustesse globale. Donc, plus il y a de relations entre les composants d'un système, plus la robustesse locale des composants peut être améliorée et plus la robustesse globale peut augmenter.

La Figure 46 montre différentes évolutions de la robustesse globale du système SMDS en considérant différentes valeurs pour l'impact des relations d'héritage. La première courbe correspond à l'hypothèse pessimiste selon laquelle les dépendance liées à l'héritage ne doivent pas être prises en compte. Pour la seconde courbe (coeff. K=0,2 pour le calcul de $Det(C_i, C_j)$), ces dépendances sont prises en compte mais de manière moins importante que les autres dépendances. Enfin, nous considérons un troisième cas, où les dépendances d'héritage sont prises en compte de la même manière que les autres. On constate sur la Figure 46 que le rôle de l'héritage est assez marginal sur l'évolution de la robustesse.

Tout ce que nous pouvons dire maintenant, c'est que les courbes la plus haute et la plus basse sont des bornes pour la véritable courbe d'évolution de la robustesse, et que de futurs travaux devraient nous permettre d'évaluer de manière plus précise l'impact des relations d'héritage pour la robustesse.

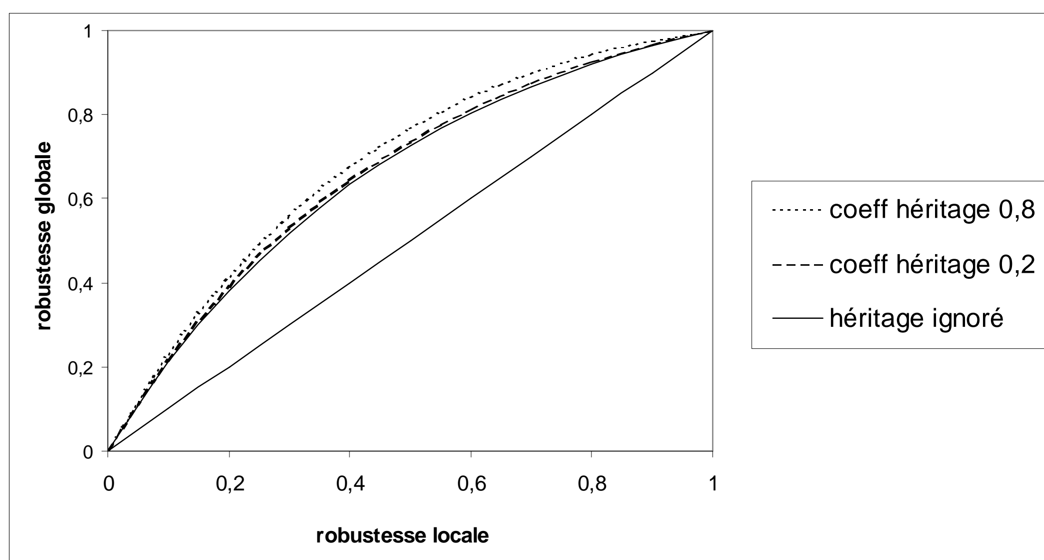


Figure 46 – Évolution de la robustesse de SMDS pour différents coefficients d'impact de l'héritage

4.2.7 Critique du modèle de mesure

Dans [Briand"02b], Briand et al. reprennent le modèle que nous proposons pour la robustesse et le généralisent en faisant l'hypothèse que tous les composants ont la même robustesse isolée (notée Rob). Les autres hypothèses et définitions sont les mêmes que les nôtres, ils considèrent notamment que $Det(C_i, C_j)$ est proportionnel à la robustesse isolée d'un composant (*i.e.*, $Det(C_i, C_j) = K \times Rob$). Ils obtiennent alors l'expression suivante pour la robustesse locale du composant i :

$RobLoc(C_i,S) = 1-(1-Rob) + (K \times Rob)^C$, où C est le nombre de clients du composant i. Le nombre moyen de clients pour les composants est ensuite fixé comme une constante dans le système, la robustesse locale est alors la même pour chaque composant. Sous l'hypothèse que chaque instruction du système a la même probabilité 1/n d'être erronée (n étant le nombre d'instruction du système), la robustesse globale s'exprime alors ainsi :

$$Rob = \sum_{i=1}^{n} (1/n \cdot RobLoc(C,S) = RobLoc .$$

En se basant sur cette généralisation du modèle, Briand et al. proposent une étude de sensibilité des paramètres C et K, dont les courbes sont données Figure 47 et Figure 48.

La principale critique que Briand émet à propos de notre modèle est que la mesure de robustesse globale est très sensible aux variations des facteurs C et K. Cependant, les courbes montrent que pour des valeurs de K entre 0,5 et 0,8 et de C entre 3 et 6 les variations sont moins importantes. Or notre hypothèse, vérifiée par toutes nos expériences, est que le facteur K se situe généralement aux alentours de 0,8. Quant à la valeur de C, il nous semble tout à fait raisonnable de faire l'hypothèse que chaque

composant a au minimum trois clients en moyenne (une étude faite dans [Hanh"02] vérifie cette hypothèse).

En conclusion, même si le modèle est très sensible aux variations de C et K, il reste tout de même valide en pratique, puisque les expériences semblent montrer que ces valeurs ne varient pas énormément, et que, dans cet intervalle, le modèle est moins sensible aux variations des valeurs que dans les intervalles extrêmes.

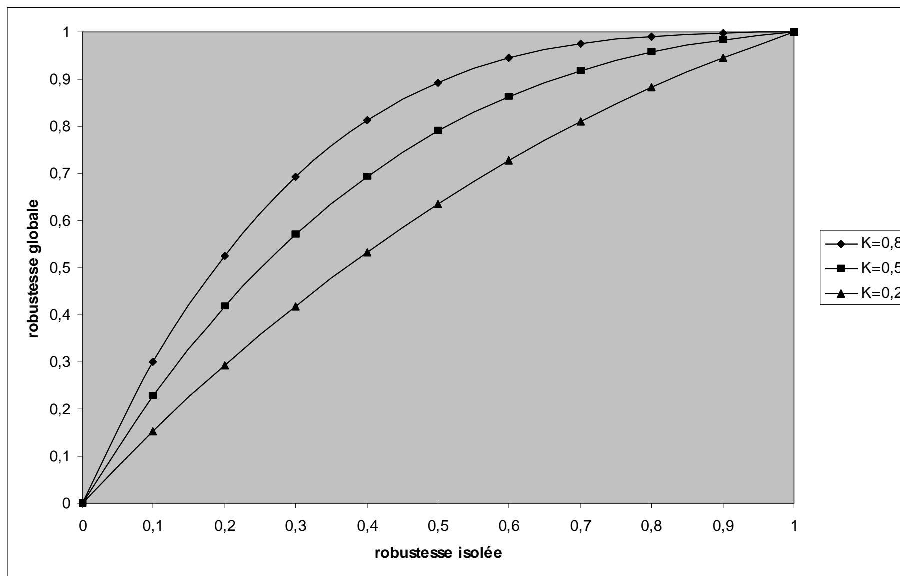


Figure 47 – Modèle généralisé par Briand : sensibilité du facteur K pour le calcul de Det(C_i, C_j) (C=3)

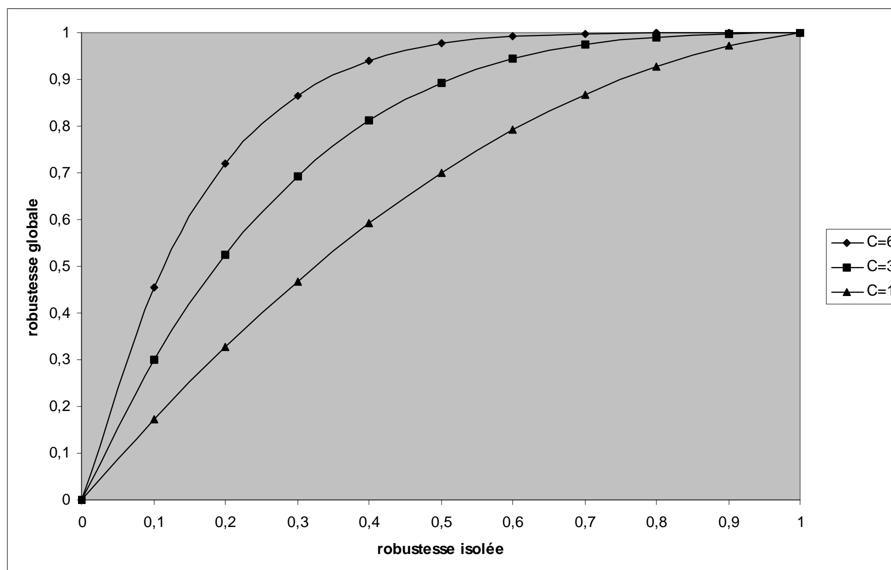


Figure 48 – Modèle généralisé de Briand : sensibilité du nombre de clients (K=0,8)

4.3 Une mesure de la diagnosabilité

La diagnosabilité est une mesure qui permet d'évaluer la précision et l'efficacité du diagnostic [Jones"02; Le Traon"03]. Comme nous l'avons vu à la section 2.1.1, lorsqu'une erreur est détectée au cours du test d'un logiciel, la phase de diagnostic consiste à localiser la faute pour pouvoir la corriger. Généralement, l'erreur n'est détectée qu'à la fin de l'exécution d'un cas de test, et, pour trouver la faute, il faut alors remonter le flot d'exécution du cas de test (l'ensemble des instructions exécutées par ce cas de test), jusqu'à trouver l'instruction erronée ou la zone suspecte (omission de code…). En présence de contrats, le diagnostic peut être facilité, puisqu'une erreur peut être détectée par un contrat qui sera alors violé. Il suffit alors de remonter le flot d'exécution avant le contrat violé, sans s'occuper des instructions après ce contrat.

L'intuition pour l'apport des contrats au diagnostic est la suivante : l'efficacité du diagnostic pour un système construit par assemblage de composants dépend de la robustesse locale des composants. En effet, plus cette robustesse locale est élevée, plus les contrats détecteront les erreurs tôt dans le flot d'exécution, plus l'erreur sera proche du contrat violé rendant ainsi le diagnostic plus efficace.

4.3.1 Analyse du problème

Avant d'analyser en détail les attributs influençant la diagnosabilité, nous rappelons les méthodes principales pour le diagnostic.

a Techniques de diagnostic pour la localisation de fautes logicielles

La première technique largement utilisée pour la localisation de fautes est fondée sur le recoupement d'informations récoltées lors de l'exécution d'un ensemble de cas de test sur un programme. Le recoupement d'informations récoltées lors du test dynamique permet alors un diagnostic semi-automatique [Jones"02]. La plupart des travaux sur le diagnostic sont fondés sur des techniques de « slicing » et se concentrent sur le test unitaire ou d'intégration. Il existe plusieurs méthodes pour le « slicing » qui consistent toutes à extraire un ensemble d'instructions qui peuvent être exécutées de manière indépendante (ce qui correspond à un slice) [Weiser"82; Weiser"84; Agrawal"95; Kamkar"95; Korel"97]. Pour localiser les fautes, il faut alors exécuter les « slices » un par un et analyser le résultat de chaque exécution. Le coût en termes d'effort humain est la principale limite de ces techniques. En effet, il faut déterminer une valeur d'oracle pour chaque « slice » ; or ils n'ont pas de fonctionnalité clairement identifiée, et l'oracle nécessite alors une intervention humaine.

Une autre technique pour la localisation des fautes consiste à insérer des assertions dans le programme pour détecter des états erronés au cours de l'exécution. Le placement systématique d'assertions au début et à la fin de chaque procédure s'avère une technique efficace pour détecter et localiser des fautes dans un programme. La conception par contrat apparaît ainsi comme une généralisation de ce principe. Cependant, l'effort nécessaire à la

définition de telles assertions peut être important : cela nécessite une bonne compréhension du fonctionnement interne des procédures ainsi que des valeurs attendues en sortie. Voas dans [Voas"99] étudie un autre type de placement des assertions fondé sur l'observabilité des variables internes.

Face au problème du diagnostic, qui reste une tâche non entièrement automatisable, il peut être utile d'avoir une estimation, a priori, de la difficulté pour la localisation des fautes. Ce facteur prédictif est appelé la *diagnosabilité* (cf. [Le Traon"98] pour la conception flot de données) et fournit aussi un facteur de qualité pour l'évaluation de la conception d'un logiciel.

Dans la suite nous explorons donc l'impact de l'utilisation de contrats sur la diagnosabilité de programmes OO.

b Diagnosabilité : analyse intuitive de la notion

L'effort pour la localisation de fautes est relatif à la taille de l'ensemble des instructions ou des composant suspects (dans lesquels on estime pouvoir trouver la faute). Plus cet ensemble est grand, plus le diagnostic est difficile. Par ailleurs, lors de la constitution de l'ensemble des instructions suspectes, on repère des sous-ensemble d'instructions non différentiables du point de vue du diagnostic (rien ne permet de suspecter une instruction plutôt qu'une autre dans un tel sous-ensemble). En plus de l'effort, on associe un critère de précision du diagnostic qui correspond à la taille maximale des sous-ensembles d'instructions non différentiables. Ces diverses considérations permettent d'exprimer les trois propositions suivantes pour la diagnosabilité :

- la diagnosabilité concerne à la fois l'effort de localisation et de précision du diagnostic,
- l'effort de localisation et la précision du diagnostic sont très fortement corrélés,
- la diagnosabilité dépend de la capacité d'une technique à isoler les instructions dans le système soit en appliquant une stratégie de test particulière, soit en insérant des assertions ou des contrats. Dans la suite de cette section nous nous concentrons sur les contrats pour isoler les instructions suspectes dans un flot d'exécution. Dans les perspectives possibles pour la suite de cette thèse (chapitre 6) nous abordons une technique fondée sur le recoupement de traces d'exécution de cas de test pour isoler les instructions suspectes.

L'attribut principal pour la localisation des instructions erronées parmi un ensemble d'instructions suspectes est appelé *indiscernabilité*. Les deux critères pour la diagnosabilité (effort et précision), sont liés au nombre d'instructions non différentiables parmi lesquelles la faute doit être trouvée.

4.3.2 Définitions

A partir des propositions précédentes nous proposons une définition de la mesure de diagnosabilité ainsi que plusieurs hypothèses pour un modèle de mesure qui sera détaillé dans la section suivante.

Diagnosabilité (définition informelle). *La diagnosabilité exprime l'effort pour la localisation d'une erreur ainsi que la précision du diagnostic pour un système précis.*

Alors que la mesure de la robustesse se concentrait sur un composant et ses contrats comme une unité d'assemblage, la diagnosabilité s'intéresse aux instructions exécutées et aux contrats traversés par un cas de test qui échoue. La notion de composant est donc mise en arrière-plan ici au profit de la notion de flot d'exécution.

Flot d'exécution. *Un flot d'exécution est un ensemble d'instructions et de contrats partiellement ordonnés qui correspondent à une exécution particulière d'un programme.*

Les instructions dans un flot d'exécution sont partiellement ordonnées puisqu'une partie de l'exécution peut être distribuée sur plusieurs processus (cf. Figure 35). Dans la suite, pour simplifier le modèle mathématique, nous ne considérons que des flots non distribués. Le modèle complet n'est pas présenté car de tels modèles de flot ne modifient pas les résultats sur la mesure de manière significative, alors qu'ils rendent le modèle mathématique très complexe.

Instructions non différentiables. *Deux instructions sont non différentiables l'une de l'autre si elles sont entourées de deux contrats consécutifs dans un flot d'exécution.*

Ensemble indiscernable. *Un ensemble indiscernable correspond à un ensemble d'instructions non différentiables.*

Dans les définitions suivantes, nous distinguons des mesures de diagnosabilité locales (au niveau d'une instruction ou d'un flot d'exécution) qui sont plus directement liées à l'effort de localisation, et une mesure de diagnosabilité globale (pour un système) liée à la précision du diagnostic.

Effort local de diagnostic (δ). *La diagnosabilité locale δ d'une instruction Inst dans un flot d'exécution F est l'effort probable pour déterminer si Inst est erronée dans F connaissant que le nombre d'instructions, le nombre et la distribution des contrats et leur efficacité.*

L'effort pour le diagnostic local dépend de l'efficacité des contrats qui correspond à la robustesse locale d'un composant, et du nombre d'instructions suspectes parmi lesquelles la faute doit être localisée.

Effort local de diagnostic pour un flot. *L'effort local de localisation d'une faute dans un flot F est l'effort probable nécessaire à la localisation de l'instruction erronée sachant qu'une faute a été détectée dans F et connaissant le nombre d'instructions, le nombre de contrats et leur efficacité.*

Diagnosabilité globale d'un système. *La diagnosabilité globale d'un système S et le degré probable de précision du diagnostic relatif à la densité et la précision des contrats dans S.*

Le modèle est fondé sur les hypothèses suivantes :

- le programme est supposé contenir au moins une faute : il existe donc une exécution du système qui provoque une défaillance si l'erreur n'est pas détectée par un contrat
- les contrats sont supposés corrects
- si un flot d'exécution contient plusieurs fautes, le diagnostic révèlera le premier point de divergence du flot avec le flot attendu (nous considérons que les fautes qui se compensent mutuellement sont négligeables)

Plusieurs paramètres pour la mesure ont été identifiés dans ces définitions (robustesse, nombre d'instructions non différentiables, flot d'exécution). La section suivante propose des définitions formelles de ces paramètres, ainsi que de la diagnosabilité à partir de ces paramètres.

4.3.3 La mesure de diagnosabilité

A partir de différents attributs identifiés précédemment, il est possible d'énoncer un modèle pour la diagnosabilité. Une fois ce modèle présenté, nous donnons quelques résultats expérimentaux sur le paramétrage des différents attributs, et sur l'évolution de la diagnosabilité d'un système en fonction de ces attributs.

a Mesure locale de la diagnosabilité

Le premier attribut que nous déterminons pour le calcul de la diagnosabilité d'un système correspond à la probabilité qu'une instruction erronée soit détectée par un contrat. Puisque nous ne distinguons pas les instructions à l'intérieur d'un ensemble indiscernable, cette probabilité est la même pour tout instruction dans un tel ensemble. Cette probabilité dépend du nombre de contrats le long du flot d'exécution, que nous appelons #contrats.

Det_i^j**.** *Det_i^j, tel que i ∈[1…#contrats] et j ∈ [i…#contrats], désigne la probabilité qu'une instruction erronée dans un ensemble indiscernable EI_i soit détectée par un contrat j, sachant qu'aucun contrat intermédiaire sur le flot n'a détecté d'erreur.*

La probabilité $\mathbf{Det}_i^j$ est contrainte de la manière suivante : $\sum_{j=i}^{\#contrats} \mathrm{Det}_i^j + P_{déf} = 1$. Ici, $P_{déf}$ désigne la probabilité de défaillance lorsque aucun contrat n'a détecté l'erreur et que la défaillance se propage jusqu'à la sortie du système. Soit p_k la probabilité que le $k^{ème}$ contrat détecte l'erreur présente dans EI_i : puisque pour un contrat j, les non-détections de l'erreur par chaque contrat précédent dans le flot d'exécution sont des évènements indépendants, on a : $\mathrm{Det}_i^j = p_j \cdot \prod_{k=i}^{j-1}(1-p_k)$.

Le second attribut auquel nous intéressons correspond à la *zone de diagnostic*, i.e. au nombre d'instructions exécutées entre l'instruction erronée et l'endroit où l'erreur est détectée. Lorsque des contrats sont présents dans le programme, la zone de diagnostic pour chaque instruction dans un ensemble indiscernable est approximativement la même.

Effort local de diagnostic (δ). *Pour un flot d'exécution particulier, l'effort local pour le diagnostic pour un ensemble indiscernable est égal à la zone de diagnostic.*

Il reste maintenant à estimer la zone de diagnostic. Si la faute est détectée par un contrat j, alors la zone de diagnostic en fonction des ensembles indiscernables EI_i est égale au nombre d'instructions entre l'instruction erronée dans EI_j et le contrat qui détecte l'erreur, soit : $\mathrm{ZoneDiagnostic} = \sum_{k=i}^{j} |EI_i|$.

Effort de diagnostic δ_i. *L'effort de diagnostic pour une instruction dans un ensemble indiscernable EI_i est :*

$$\delta_i = \sum_{j=i}^{\#contrat} \mathrm{ZoneDiagnostic}_i \times \mathrm{Det}_i^j + P_{déf} \times \mathrm{ZoneDiagnostic}_{\#contrats}$$

, où la probabilité $P_{déf}$ s'obtient de la manière suivante : $P_{déf} = 1 - \sum_{j=i}^{\#contrat} \mathrm{Det}_i^j$.

Le dernier terme dans le calcul de δ_i est ajouté pour mesurer la zone de diagnostic lorsque aucun ne détecte l'instruction erronée (l'erreur se propage jusqu'à la sortie du système, provoquant ainsi une défaillance). L'effort de diagnostic pour un flot correspond à la zone de diagnostic probable connaissant la probabilités $Perr_i$ que l'erreur soit dans EI_i. On obtient donc l'effort de diagnostic comme suit : $\delta = \sum_{i=1}^{\#contrats} (Perr_i \times \delta_i)$

Les différentes formules énoncées ci-dessus pour le calcul de l'effort de diagnostic sont trop générales pour une application pratique de la mesure pour un flot. Nous définissons donc les hypothèses suivantes qui permettent de simplifier considérablement le modèle :

1. les contrats sont répartis uniformément dans un flot d'exécution. Chaque ensemble indiscernable a donc la même taille tailleEI qui est égale au nombre d'instructions divisé par le nombre de contrats sur le flot (hypothèse justifiée plus loin)
2. plus un contrat est proche de l'instruction erronée, plus la probabilité qu'il a de détecter la faute est grande (i.e. p_k décroît lorsque k augmente, $k \in [i \ldots \#contrats]$)
3. les contrats ont tous la même probabilité de détecter une faute dans une instruction les précédant immédiatement
4. chaque instruction a une probabilité uniforme d'être erronée égale à 1/#instr.

Des expériences ont été réalisées pour justifier la validité de la première hypothèse. Comme nous avons réalisé ces expériences sur des programmes écrits en Java, ne contenant pas tous des contrats, la répartition des contrats est estimée avec la répartition des appels de méthode sur un flot. En effet, si nous faisons l'hypothèse qu'il y a un contrat au début et à la fin de chaque méthode (ce sera vrai s'il y a au moins un invariant pour la classe), alors la répartition des appels de méthode correspond effectivement à une approximation de la répartition des contrats dans un flot. Nous avons mesuré cette répartition sur cinq systèmes de taille différentes. La taille des flots d'exécution va de 1400 instructions à presque deux millions. Le Tableau 6 détaille les caractéristiques de chaque système. Il apparaît que les contrats sont uniformément répartis dans ces systèmes et que la première hypothèse est valide et peut donc être prise en compte pour le modèle de la diagnosabilité. Le détail des courbes pour ces mesures est donné en Annexe A.

Tableau 6 – Répartition des contrats

	#instructions	#contrats	#instructions / #contrats
classe List	1391	505	2,75
Serveur de réunions virtuelles	2291	1171	1,96
Auto-Test pour JUnit	19419	10801	1,8
JDK	111730	40751	2,74
JTree	1970745	885001	2,22

Nous avons développé un outil spécifique pour obtenir ces données sur la répartition des contrats dans un flot d'exécution. Cet outil est baptisé JTracor, et permet de tracer l'exécution d'un programme JAVA en utilisant les facilités qu'offre l'interface JDI [SUN"02] (Java Debugging Interface). Cette interface offre un système de requêtes sur les évènements de la machine virtuelle (appel de méthode, instruction exécutée…), et permet ainsi de tracer l'exécution d'un programme sans instrumenter le code.

La seconde hypothèse correspond à l'intuition et a été vérifiée expérimentalement dans la section précédente. La troisième hypothèse (ainsi que la première) permet de proposer une mesure simplifiée qui évalue l'impact des contrats pour la diminution de l'effort de diagnostic. L'efficacité de chaque contrat pourrait être mesurée par analyse de mutation comme nous l'avons décrit à la section 4.2.5, mais ce n'est pas l'objet de l'étude qui cherche à extraire des résultats généraux. Enfin, la quatrième hypothèse est raisonnable dans une approche prédictive puisqu'elle force à suspecter chaque partie du programme de la même manière.

Pour modéliser la seconde hypothèse et rester cohérent avec la troisième, nous considérons que le premier contrat exécuté immédiatement après l'instruction erronée a une probabilité p de détecter l'erreur, le second une probabilité $\alpha.p$, le troisième $\alpha^2.p$ et ainsi de suite. La constante α est appelée le *coefficient d'absorption* et est compris entre 0 et 1. Lorsque ce coefficient est égal à 1 tous les contrats sont équivalents et la seconde hypothèse ne peut pas être vérifiée, ce qui correspond au modèle le plus optimiste. Si le coefficient est égal à 0, cela signifie que seul le premier contrat peut détecter l'erreur.

A partir de ce coefficient, nous pouvons exprimer la probabilité de détection suivante : $Det_i^j = \alpha^j p \cdot \prod_{k=i}^{j-1}(1 - p_{k-i})$. La zone de diagnostic peut se calculer ainsi

ZoneDiagnostic = (#contrats – i +1)* tailleEI. Enfin l'effort local de diagnostic s'exprime ainsi :

$$\delta_i = tailleEI \cdot [\sum_{j=i}^{\#contrats}(j-i+1) \cdot Det_i^j + (1 - \sum_{j=i}^{\#contrats} Det_i^j) \times (\#contrats - i + 1)].$$

b Paramétrage du modèle

L'estimation de la qualité des contrats est détaillée à la section 4.2.5, et révèle que cette qualité varie de 0,17 à 1 (entre 17% et 100% de détection d'erreur par les contrats), et qu'une valeur moyenne raisonnable se situe autour de 0,8.

Il faut ensuite paramétrer le coefficient d'absorption. La Figure 49 décrit une analyse de sensibilité pour ce coefficient. Par exemple, pour des contrats d'efficacité 0,4, le nombre probable de contrats traversés avant la détection de l'erreur se situe entre 3,1 et 6,1 pour dix contrats maximum. On peut aussi observer sur cette figure qu'une mauvaise estimation du coefficient d'absorption peut induire une perte d'information lors du calcul de la diagnosabilité globale (en particulier si l'efficacité des contrats se situe entre 0,2 et 0,7). cependant, nous considérons que les différences restent acceptables et qu'elles ne seront jamais en contradiction avec les axiomes.

Les expériences réalisées sur la librairie Pylon ont permis d'évaluer le coefficient d'absorption à 0,8.

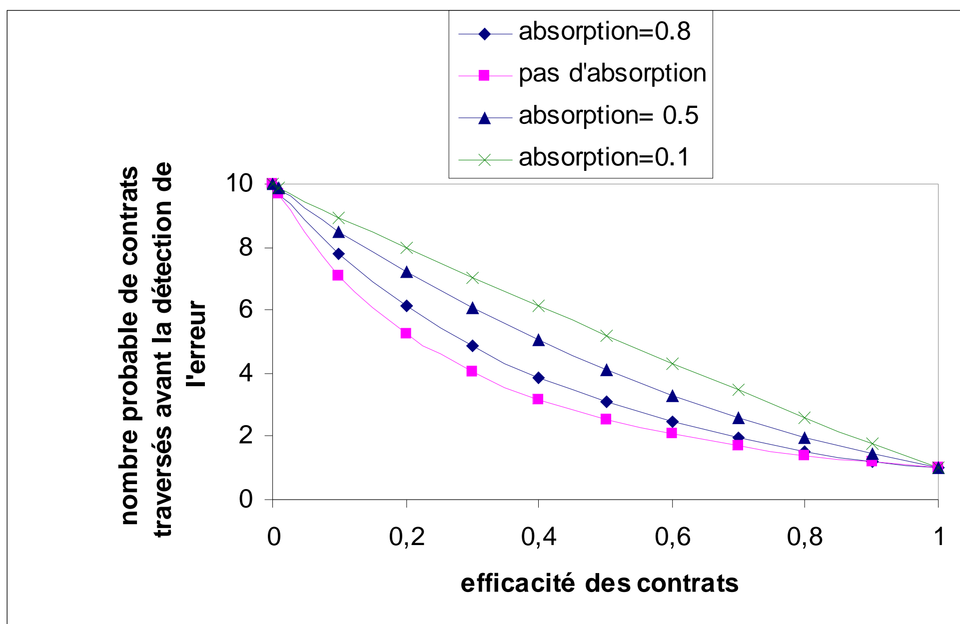


Figure 49 – Paramétrage pour 10 contrats

c Résultats des mesures de la diagnosabilité globale

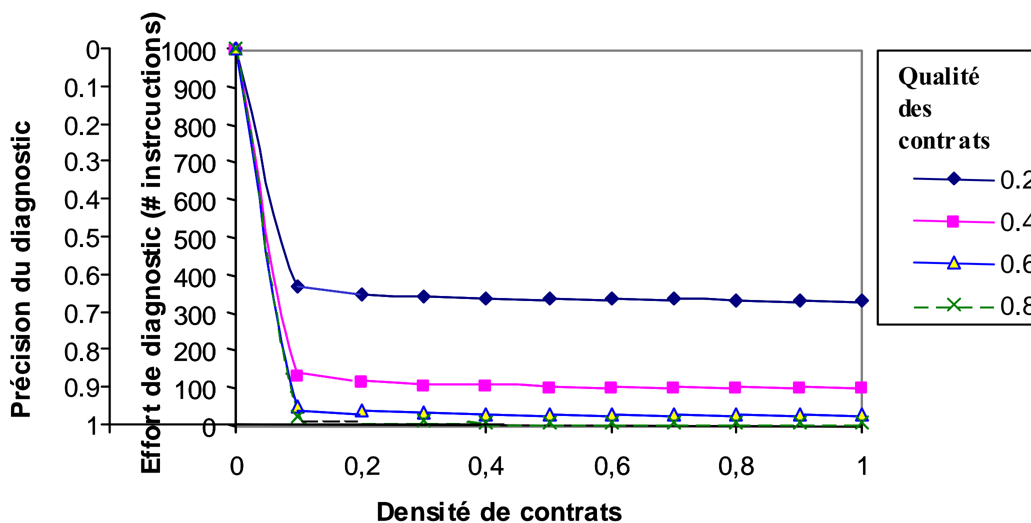


Figure 50 – Résultats pour δ et Δ

Comme nous l'avons exprimé plus tôt, l'effort de diagnostic pour un flot d'exécution est proportionnel à la taille de ce flot, tous les paramètres étant fixés par ailleurs (efficacité et densité des contrats). Ceci peut être démontré facilement à partir du fait qu'une telle courbe d'effort est une fonction homothétique centrée en 1 avec un coefficient multiplicatif égal à la

taille du flot. Il apparaît alors que la précision du diagnostic ne dépend pas de la taille du flot, ce qui donne à ce résultat un caractère général.

La Figure 50 représente les même résultats sur deux échelles différentes en termes d'effort « absolu » pour le diagnostic et en terme de précision relative. Les courbes pour l'effort de diagnostic sont données pour différentes valeurs pour l'efficacité des contrats et sur un flot de 1000 instructions, le coefficient d'absorption étant égal à 0,8.

La diagnosabilité globale s'obtient donc directement pour n'importe quel flot d'exécution, i.e. elle ne dépend que de l'efficacité et de la densité des contrats.

4.3.4 Résultats et conclusions

Sur la Figure 50, nous remarquons tout d'abord que le simple fait d'introduire des contrats améliore rapidement la diagnosabilité du système. Ensuite, l'ajout de nombreux contrats (grande densité de contrats) n'augmente pas la diagnosabilité de manière significative (qui est bornée par 0,6 dans le cas de contrats avec une efficacité de 0,2 ou par 0,9 dans le cas de contrats avec une efficacité de 0,4). Enfin, la qualité des contrats est plus importante que leur quantité puisque c'est le seul moyen d'augmenter la borne maximale de la diagnosabilité globale.

Nous pouvons déduire plusieurs choses de ces observations :

- une densité de 0,2 (*i.e.*, 2 contrats pour 10 instructions) pour les contrats est suffisante pour atteindre la valeur maximale de la diagnosabilité, la valeur de l'efficacité des contrats étant fixée par ailleurs. Remarquons que dans les programmes OO, les méthodes sont souvent petites et qu'une densité de 0,2 pour les contrats est facilement atteignable dans ce cas. Il est donc inutile de rajouter des assertions à l'intérieur des méthodes. Remarquons aussi que ce résultat était difficilement prévisible sans modèle mathématique.
- La qualité est plus importante que la quantité puisque, pour une densité fixée, seule la qualité des contrats permet de changer la valeur de la diagnosabilité. Il est donc plus important de fournir un effort pour une bonne conception et des interfaces bien définies (permettant de définir clairement des propriétés sous forme de contrat), plutôt que de fournir un effort sur des assertions défensives, qui sont souvent imprécises et dépendantes du code.

La conception par contrat apparaît donc comme une technique efficace pour l'amélioration de la diagnosabilité d'un système vu comme un assemblage de composants, et, de manière plus générale, de la qualité globale de ce système.

4.4 Conclusion

Au cours de cette étude nous avons pu estimer l'impact réel des contrats sur deux facteurs de qualité que sont la robustesse et la diagnosabilité. Ces études, qui mériteraient d'être

complétées, montrent que la simple introduction des contrats améliore de manière importante ces deux facteurs.

Ce travail, a débuté par la présentation d'un cadre générique pour la définition d'une mesure. Les deux mesures ont ensuite été définies dans ce cadre. Les paramètres significatifs ont été isolés pour chaque mesure, et un modèle a été défini en fonction de ces critères. Des expériences ont permis de fixer les valeurs pour les paramètres, et d'observer le comportement des mesures en fonction de ces valeurs. Les résultats ont montré que la simple introduction de contrats augmente rapidement la robustesse et la diagnosabilité. Dans le cas particulier de la diagnosabilité, au-delà d'une densité minimale, la qualité des contrats est le facteur le plus important pour ces deux mesures.

Les résultats pour l'étude de la mesure de robustesse ont aussi montré la limite des contrats comme fonction d'oracle pour le test. Cependant, nous pensons qu'une étude approfondie devrait permettre d'établir des règles pour classer les cas de test en fonction de la probabilité qu'une erreur soit détectée par les contrats en exécutant le cas de test. Il faudrait alors écrire un oracle explicite en priorité pour les cas de test qui ont une faible probabilité de détecter des erreurs avec les contrats.

5
Anti-patterns de testabilité dans un assemblage de composants

Un principe important de la conception orientée objet consiste à répartir le traitement d'une fonctionnalité dans de nombreux objets du système. En conséquence, le contrôle est réparti sur plusieurs objets, et le test doit stimuler, soit explicitement, soit de façon indirecte les interactions entre objets. De plus, puisque les collaborations entre objets sont cruciales, il est également crucial de les détecter tôt et de déterminer la difficulté pour les couvrir lors du test. Au cours de ce chapitre, nous proposons donc un critère de test pour couvrir les interactions entre objets. Celui-ci nous permet ensuite d'étudier la *testabilité* [Binder"94; Voas"95] des modèles statiques orientés objet à partir des diagrammes de classes UML(Unified Modelling Language). Cette mesure prédictive permet de détecter les interactions potentielles sur le diagramme et de désigner précisément les zones à améliorer pour diminuer l'effort de test (évalué en fonction du critère proposé ici).

5.1 Présentation de la problématique

Quand un modèle OO est mal spécifié ou ambigu, des interactions d'objets non souhaitées sont également implicitement spécifiées : toute méthode de génération de test, fondée sur la couverture d'interactions d'objets, produit alors des cas de test inutiles (qui cherchent à couvrir des interactions infaisables). De tels objectifs de test sont conformes à la spécification, mais ne semblent plus raisonnables au moment d'être appliqués sur l'implantation. L'objectif de l'analyse de testabilité, au niveau d'un assemblage de composants, est d'indiquer les parties de cet assemblage où des problèmes dus aux interactions peu testables doivent être examinés. De telles configurations peu testables dans l'assemblage sont appelées *anti-patterns de testabilité* dans cette thèse par analogie avec les anti-patterns de conception répertoriés dans [Brown"98]. Un anti-pattern décrit une solution à un problème récurrent qui produit des conséquences négatives sur la qualité du logiciel

[Correa"00]. Or, les configurations que nous détectons ici sont récurrentes et peuvent dégrader la testabilité de l'assemblage de composant.

Pour guider la tâche de test, la principale vue statique d'une conception OO, à savoir le diagramme de classes, semble fournir une bonne base pour détecter et maîtriser les dépendances implicites de contrôle réparties dans le système, qu'elles soient dues à l'héritage ou à la liaison dynamique. Cependant, un diagramme de classes est souvent ambigu, inachevé, et peut mener à plusieurs interprétations fausses, se dérivant en des implantations erronées et, incidemment, à un nombre prohibitif d'objectifs de test cherchant à couvrir ces points d'ambiguïté. Une analyse de testabilité doit donc permettre, autant que possible, de détecter ces ambiguïtés. On pourrait arguer que les vues complémentaires d'UML, telles que les diagrammes de séquence, de collaborations ou d'objets servent précisément à lever certaines ambiguïtés : ceci n'est pas vrai pour le test. En effet, elles n'offrent que des « instantanés » de configurations valides et ne permettent pas d'éliminer les comportements non souhaités. Dans la cas du test, le rôle des diagrammes de collaborations se restreint à l'expression des traces qu'un cas de test doit produire [Abdurazik"00], de même que les diagrammes de séquence offrent une base pour indiquer des objectifs nominaux et exceptionnels de test [Pickin"02]. Les diagrammes de collaborations et de séquence peuvent aider à comprendre les interactions mais ne peuvent pas détailler chacune d'entre elles, ni limiter leur nombre. De la même manière, les diagrammes d'objets représentent seulement une configuration particulière des instances de classes dans le système, mais ne décrivent pas toutes les configurations valides. Seuls les diagrammes d'états représentent exhaustivement un comportement dynamique donné, mais on ne dispose pas nécessairement de cette vue pour toutes les classes du système. De même, leur combinaison – du fait de la création dynamique d'objets et de la communication asynchrone entre objets – produit des automates infinis ou de taille telle qu'on ne peut pas toujours les traiter. En conséquence, nous considérons que les vues principales sur lesquelles la testabilité doit être analysée sont les diagrammes de classes et d'états, et que les autres vues montrent seulement des instantanés de quelques comportements possibles. L'étude décrite dans ce chapitre se concentre sur les diagrammes de classes UML.

Les problèmes de testabilité dans un diagramme de classes sont dus à l'existence de relations client/serveur dans le système, tout comme pour n'importe quel logiciel classique. En effet, s'il n'y a aucun client dans le logiciel, il n'y a pas non plus d'ensemble défini d'exécutions, et donc il n'y a rien à tester. Aussi, après le test unitaire, les défaillances devraient se produire seulement en raison d'interactions erronées entre objets : ces interactions traversent toute l'architecture et sont d'autant plus complexes lorsque les dépendances client/serveur traversent des arbres d'héritage. Les dépendances polymorphes multiplient le nombre potentiel de types d'objet qui peuvent agir les uns sur les autres avec diverses – et peut-être fausses – implémentations. Un critère de test qui exige la couverture explicite de ces interactions d'objets doit alors être défini. Pour être applicable, le nombre de cas de test doit rester raisonnable et nous proposons une évaluation de l'effort de test, mesuré

en estimant le nombre d'interactions d'objets à partir du diagramme de classes d'UML. L'évaluation que nous considérons comme significative de la testabilité globale d'un diagramme de classes est donc le nombre « d'interactions de classes ». Une interaction de classes est une configuration topologique dans le diagramme de classes que les tests doivent couvrir. Elle se produit si une classe est fournisseur d'autres classes par divers chemins possibles de dépendances.

A partir du critère de test proposé, notre objectif est triple :

- Fournir un modèle qui capture les interactions de classes et indique exactement les causes de ces interactions. On appelle ces configurations recouvertes dans le diagramme de classes des anti-patterns de testabilité. Dans la suite nous en définissons deux.
- Mesurer le nombre et la complexité (due au polymorphisme) des anti-patterns. Cette mesure est considérée ici comme notre évaluation de testabilité d'un modèle.
- Suggérer des améliorations au modèle pour réduire le nombre et la complexité des anti-patterns de classes : ces améliorations au niveau du modèle sont réalistes puisque les vérifications statiques sur le code assurent leur exécution.

Quand ces objectifs sont atteints, nous pouvons rejeter ou accepter un modèle selon un critère de testabilité. On peut alors choisir entre rejeter le modèle si aucune amélioration ne peut limiter les anti-patterns, ou bien passer outre, et accepter de ne pouvoir couvrir toutes les interactions. Le modèle doit être assez précis pour repérer exhaustivement les interactions, en restant assez simple, et proche du contexte réel, pour aider à résoudre les problèmes de testabilité.

Une fois que le modèle et le critère de test sont définis pour un diagramme de classes, nous étudions la testabilité des design patterns [Gamma"95]. Deux raisons nous ont conduits a à étudier les design patterns dans le contexte de la testabilité. D'abord, nous montrons que l'application de patrons aide à résoudre des problèmes classiques de testabilité en diminuant la taille des méthodes et en supprimant les structures explicites de contrôle. L'inconvénient est que de nombreuses classes et relations supplémentaires sont introduites, conduisant à des interactions entre ces classes. L'application de design patterns semble donc simplifier la structure du code, au prix d'un modèle plus complexe. Nous appliquons ainsi notre analyse de testabilité sur des applications de design patterns et nous proposons des solutions pour résoudre les problèmes qui sont soulevés à la fin de l'analyse.

La deuxième raison pour laquelle les design patterns sont étudiés ici, est qu'ils offrent une décomposition de grands diagrammes de classes en sous-ensembles cohérents. Certaines méthodologies aident à établir une architecture OO en raffinant un premier diagramme de classes par l'application des design patterns. Ces étapes d'amélioration peuvent être vues comme la cristallisation des microarchitectures autour d'un premier diagramme de classes

central. En utilisant une telle méthodologie, les problèmes de testabilité peuvent être plus facilement identifiés et résolus au niveau du sous-ensemble.

La section 5.3 présente tout d'abord notre critère de test pour couvrir les interactions de classes. Nous y présentons également de manière informelle deux anti-patterns qui peuvent mener à des modèles difficiles à tester. La section 5.4 présente un modèle de graphe qui fournit une abstraction d'un diagramme de classes appropriée à l'analyse de testabilité. Enfin, la section 5.5 donne des indices pour préciser le modèle afin de diminuer le nombre d'interactions de classes. Les sections 5.7 et 5.8 traitent exclusivement des architectures fondées sur des design patterns.

5.2 Testabilité d'une architecture OO: définitions et méthodologie

Cette section présente le contexte général pour notre étude de la testabilité des diagrammes de classes UML. Nous revenons tout d'abord sur l'approche globale pour la construction d'une mesure de testabilité dans le cadre particulier de l'étude présentée ici. A partir de cette mesure, il est alors possible de définir une méthodologie pour la conception de systèmes testables. Enfin, nous présentons un exemple simple qui sera utilisé pour illustrer certaines notions au cours de cette étude sur la testabilité.

5.2.1 Testabilité

Le facteur de testabilité apparaît tout d'abord comme abstrait et difficile à estimer. En effet, comment évaluer la facilité pour tester un logiciel ? La solution fréquemment employée consiste à considérer un autre facteur plus facilement quantifiable comme un bon indicateur de la testabilité. Comme nous l'avons abordé à la section 2.6, au cours de ces travaux nous nous concentrons sur l'effort de test.

Nous avons eu une approche pragmatique de cette mesure, contrairement à l'étude sur la robustesse et la diagnosabilité (chapitre 4), où nous proposions une axiomatisation de la mesure, conformément aux approches 'top-down' classiques pour l'élaboration de mesures. Dans le cas de l'étude de la testabilité des diagrammes de classes, nous avons d'abord étudié précisément des applications concrètes pour identifier les attributs significatifs de la testabilité. Le travail préliminaire à cette étude consiste à définir précisément le critère de test que nous cherchons à satisfaire, et à être capable d'évaluer l'effort nécessaire pour tester un programme en fonction de ce critère.

Le critère que nous avons retenu comme révélateur de la faiblesse d'une architecture pour la testabilité, est la notion d'*anti-pattern de testabilité*. De la même manière que les design patterns correspondent à de bonnes solutions de conception dans un cadre particulier, les anti-patterns correspondent à de mauvaises solutions, qui peuvent endommager la qualité d'une conception pour un critère particulier. Nous identifions deux anti-patterns pour la testabilité, qui correspondent à des configurations dans le diagramme de classes qui peuvent entraîner

une implantation difficile à tester. Dans la suite de ce chapitre nous les définissons précisément, ce qui nous permet d'expliciter le critère de test que nous cherchons à vérifier. Une mesure de complexité est ensuite associée à ce critère de test, qui correspond à l'estimation de l'effort pour vérifier le critère, et donc à l'estimation du coût de test.

5.2.2 Une méthodologie pour la conception de systèmes testables

La

Figure 51 présente l'organigramme de la méthodologie que nous préconisons pour améliorer la testabilité d'une architecture, applicable dans un cadre OO. La première étape de cette méthode consiste à faire une analyse de testabilité pour un diagramme de classes. Cette analyse permet de repérer des points dans la conception qui sont la cause d'une faiblesse de l'architecture du point de vue de la testabilité. Comme cela est décrit dans la section 5.3, ces points correspondent à des configurations particulières dans le diagramme de classes qui peuvent conduire à des implantations difficiles à tester, que nous appelons des anti-patterns de testabilité. Pour pouvoir analyser un diagramme automatiquement, il est nécessaire de le modéliser de manière non ambiguë pour appliquer une analyse des points critiques pour la testabilité.

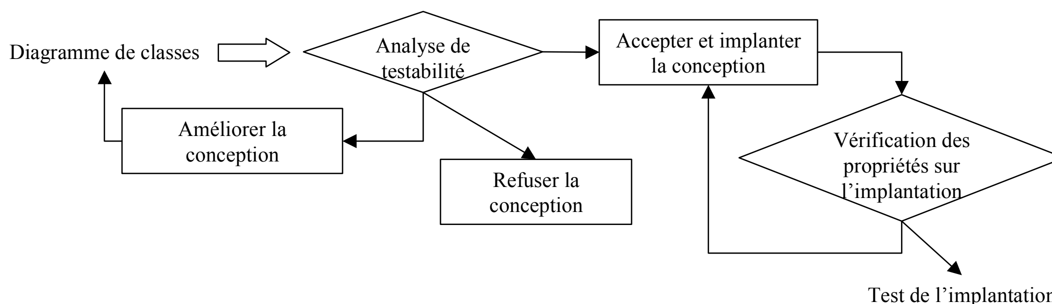


Figure 51 – Méthode pour améliorer la testabilité d'un logiciel OO

L'analyse détermine les points du diagramme de classes qu'il faut améliorer. Cette analyse permet également d'obtenir une mesure de la complexité de ces points critiques pour la testabilité (cf. section 5.4.3). Une fois cette analyse effectuée, la décision doit être prise entre améliorer ou rejeter le diagramme de classes s'il est jugé trop difficile à tester ou bien l'accepter et l'implanter dans le cas contraire. Une manière d'améliorer la testabilité du diagramme peut être de réduire le couplage [Fowler"01] ou d'exprimer des contraintes qui aideront le développeur à éviter d'implanter des interactions entre objets difficiles à tester. Dans la section 5.4.3 nous suggérons une autre piste, moins coûteuse, qui consiste à utiliser des stéréotypes sur les associations et les dépendances pour spécifier le rôle de ces liens (création, lecture ou écriture). Quand le logiciel est implanté, les contraintes correspondant aux stéréotypes sont vérifiées.

5.2.3 Exemple

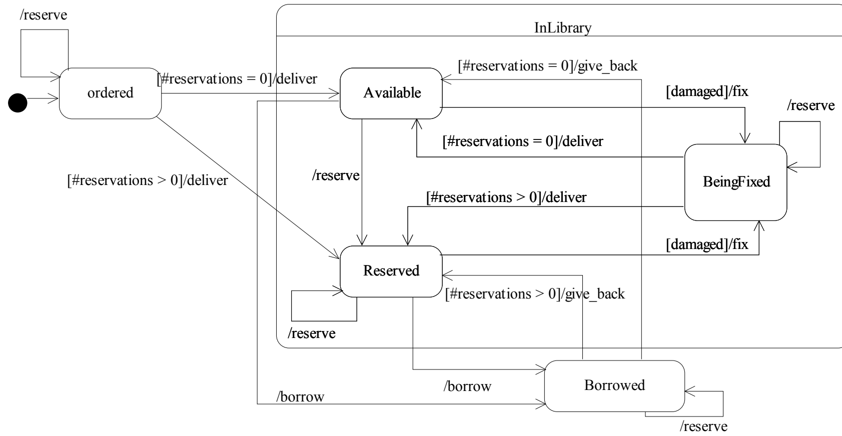


Figure 52 – Diagramme d'états pour le gestionnaire de livre

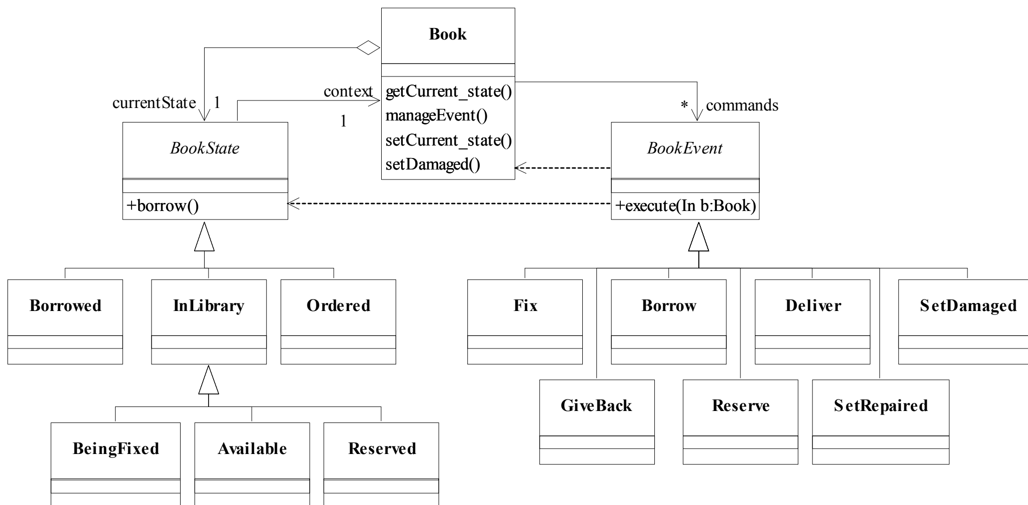


Figure 53 – Diagramme de classes UML pour un sous-système de gestion d'un livre

Nous présentons ici l'exemple qui illustre certains points tout au long de ce chapitre sur la testabilité. Cet exemple consiste en un sous-système pour gérer un livre dans une bibliothèque. Le diagramme d'états de la Figure 52 décrit le comportement dynamique d'un objet livre. Un objet est créé lorsqu'un livre est commandé (état initial), il peut ensuite être réservé à tout moment. Quand le livre arrive à la bibliothèque, il est soit disponible, soit réservé, et il peut alors être emprunté. Si le livre est endommagé et qu'il est dans la bibliothèque, il peut être réparé.

La Figure 53 illustre un diagramme de classes possible pour implanter la machine d'états. Dans ce diagramme de conception les états et les évènements de la machine d'états ont été

réifiés en appliquant les design patterns State et Command. Toutes les classes, associations et dépendances sont données, mais, pour une raison de lisibilité, ne sont visibles que les méthodes et attributs qui sont utilisés pour les exemples.

Le contexte général pour l'étude de testabilité étant établi, la section suivante décrit les types d'interactions qui sont analysées pour estimer la testabilité d'un diagramme de classes. Nous proposons ensuite un critère de test pour couvrir ces interactions, et nous illustrons ce point en écrivant les cas de test nécessaires à la vérification du critère pour le sous-système de gestion de livre.

5.3 Critère de test et anti-patterns pour les architectures OO

L'intérêt d'un critère de test est d'offrir un moyen objectif de contrôler l'arrêt de la génération de cas de test, et un objectif à atteindre pour les testeurs. Dans le cadre des interactions d'objets, il est utile d'offrir un critère qui formalise ce que les cas de test doivent couvrir dans un diagramme de classes d'un point de vue structurel. Nous proposons donc un critère qui vise à couvrir toutes les interactions entre objets susceptibles d'entraîner des effets de bord. Le diagramme de classes UML est la spécification principale utilisée pour définir ce critère. Dans la suite de cette section, nous montrons que, pour appliquer ce critère de manière raisonnable, l'architecture doit être aussi proche que possible de l'implantation.

Dans la suite, nous commençons par une description informelle des problèmes de testabilité en étudiant le diagramme de classes de la Figure 53. Ces problèmes correspondent à des configurations particulières qui apparaissent dans un diagramme de classes et qui peuvent entraîner des problèmes pour le test. Ces configurations sont appelées des anti-patterns pour la testabilité, puisqu'elles correspondent à des configurations récurrentes qu'il faut éviter pour une meilleure testabilité. Nous formalisons cette notion plus loin dans la section, et montrons aussi comment l'héritage peut augmenter la complexité de ces anti-patterns.

Ensuite, nous décrivons ces anti-patterns plus précisément en termes d'éléments dans un diagramme de classes UML, ce qui permet de définir un critère de test pour couvrir ces interactions non-désirées. Cette section se termine par un retour sur l'exemple de gestion de livre avec l'écriture des cas de test pour ce système qui vérifient le critère de test.

5.3.1 Analyse informelle des anti-patterns de testabilité

Nous décrivons ici, de manière informelle, les interactions qui nous intéressent pour évaluer la testabilité d'un diagramme de classes. Prenons l'exemple du diagramme de la Figure 53, qui correspond à un exemple typique d'architecture OO. En effet, les mécanismes orientés objet de base tels que l'héritage, les classes abstraites, et la délégation sont utilisés. Au premier regard, cette architecture révèle une forte inter-dépendabilité pour le traitement des différentes classes. Par exemple toutes les classes filles dépendent fortement de leurs

parents ou encore BOOK et BOOKSTATE dépendent l'une de l'autre. D'autre part, des dépendances indirectes peuvent aussi exister, comme entre BOOKEVENT et BOOK à travers deux chemins différents : un direct entre les deux classes et un autre via BOOKSTATE. Du fait de ces relations nombreuses et complexes, il y a un grand risque que des erreurs apparaissent au moment de l'implantation de cette architecture (de fait, des erreurs ont effectivement été commises par le programmeur, pourtant expérimenté, qui a implanté ce programme). Les deux sources de problème identifiables sur un diagramme de classes sont les suivantes :

- Quand une méthode `m1` dans la classe BOOK utilise une méthode `m` de la classe BOOKSTATE, la classe BOOKSTATE peut utiliser la classe BOOK pour traiter `m`. Cela signifie que la classe BOOK peut s'employer elle-même quand elle utilise BOOKSTATE.
- Quand la classe BOOKEVENT s'utilise elle-même, elle peut le faire de deux manières différentes: directement avec BOOK en paramètre d'une de ses méthodes ou bien via BOOKSTATE qui emploie BOOK.

Le nombre exact d'interactions, ainsi que leur complexité, est difficile à déterminer par une observation simple du diagramme, c'est pourquoi nous avons besoin d'un modèle pour les déterminer toutes de manière exhaustive.

Cette analyse informelle met en évidence deux constructions récurrentes pouvant altérer la testabilité : des interactions d'une classe à l'autre, et une configuration que nous appelons *auto utilisation*, qui correspond à une classe qui s'emploie elle-même par le biais des dépendances transitives d'utilisation.

Remarque : l'interaction d'auto-utilisation semble n'être qu'un simple « call-back ». Cependant, plusieurs raisons nous ont amené à définir et à utiliser la notion d'auto-utilisation plutôt que celle de « call-back » pour désigner l'anti-pattern de testabilité, et tout d'abord le manque d'une définition précise du « call-back » (n'importe quel appel de méthode sur l'objet appelant pour certains, appel de méthodes pour répondre aux évènements sur l'IHM pour d'autres…). Une seconde raison tient au fait que, comme nous le verrons par la suite, nous distinguons l'interaction de classes (et donc potentiel, puisqu'il est détecté sur un diagramme de classes) et l'interaction d'objets (et donc effectif puisqu'il concerne des relations entre objets du système). Or, un « call-back » ne concerne que des appels de méthodes effectifs entre objets du système. Enfin, la dernière raison, et la plus importante, est l'interprétation que nous faisons de cette boucle d'appels de méthodes. Le « call-back » insiste sur le fait qu'un objet qui est appelé par objet distant rappelle cet objet. Or, ce qui nous intéresse dans le cas de l'anti-pattern de testabilité, tient plus au fait qu'un objet qui appelle une méthode peut déclencher un appel de méthode sur lui-même et donc entraîner une modification de son propre état. Dans ce cas, la dénomination auto-utilisation nous semble donc plus appropriée pour désigner l'anti-pattern.

Ces interactions peuvent être vues comme des anti-patterns pour la testabilité, par analogie avec les anti-patterns de conception, en ce sens qu'elles correspondent à des configurations

récurrentes sur un diagramme de classes qui peuvent avoir des conséquences négatives sur la testabilité du logiciel.

Anti-pattern pour la Testabilité. *Un anti-pattern représente une mauvaise solution pour le développement d'un logiciel dans un contexte particulier. Cela correspond à une configuration dans le diagramme de classes qui peut entraîner une augmentation de l'effort de test nécessaire à la vérification du critère de test.*

La complexité des deux anti-patterns augmente lorsque les relations de dépendance traversent un arbre d'héritage en raison du polymorphisme. La section suivante illustre ce point.

5.3.2 Complexité de l'héritage

La complexité due à l'héritage apparaît quand les dépendances transitives passent par une ou plusieurs hiérarchies d'héritage. Sur la Figure 54, il y a une interaction de classes de C à D. L'interaction est complexe car si C utilise une instance de la classe A ou A2 ou A21, ces trois classes ont toujours des relations potentielles entre elles. En effet, le code de A2 peut dépendre de A comme de A21. Dans ce cas, l'interaction de C avec chacune des trois utilisations potentielles (A ou A2 ou A21) doit être examinée, et pour chacune d'elles, nous devons examiner les relations entre les classes dans la hiérarchie de l'héritage. Cependant, en contraignant le modèle (et en le rendant plus précis), nous pouvons réduire la complexité de l'interaction. En effet, si les classes A et A2 sont des classes d'interface, nous pouvons nous assurer que C ne peut effectivement utiliser que A21 ou A22, les classes parentes n'offrant plus aucun code à ces deux classes : la surface d'interaction avec la classe D est ainsi réduite à la classe A21.

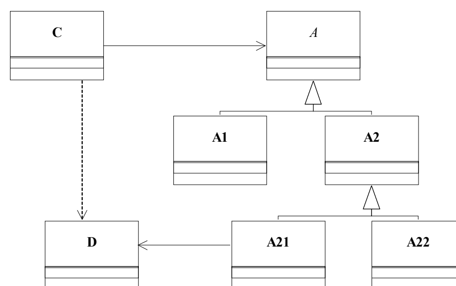


Figure 54 – Interaction de classes à travers une hiérarchie d'héritage

On obtient donc les contraintes suivantes sur les informations que notre modèle doit capturer depuis le diagramme de classes. Le modèle de test devra distinguer les dépendances orientées vers les ancêtres de celles orientées vers les descendants dans un arbre d'héritage. De même, le modèle de test ne doit pas créer de dépendances entre les classes sœurs,

puisqu'elles sont toujours indépendantes du point de vue du test. Enfin, le modèle doit également saisir la complexité de l'interaction.

5.3.3 Critère de test pour des systèmes OO

Nous revenons ici sur les anti-patterns identifiés dans la section 5.3 et en donnons une définition plus précise en termes d'éléments d'un diagramme de classes UML. Le critère de test pour couvrir ces anti-patterns lors du test de l'implantation est alors défini. Il vise à identifier les erreurs dues à des effets de bord lorsque les relations entre objets sont complexes. Par exemple, si un objet peut modifier l'état d'un autre de manière indirecte à travers plusieurs chemins de dépendance différents, l'état de l'objet modifié peut devenir incohérent vis-à-vis de l'objet client. Dans un tel cas, des test indépendants pour chaque relation ne permettront pas de détecter cette incohérence : il faudra tester cette interaction explicitement avec un unique scénario de test.

Dans un système OO, les classes dépendent les unes des autres pour leur traitement. Une classe a utilise une classe b si des méthodes de la classe a font appel à des méthodes de b, sur un attribut ou une variable locale de type b. Ces relations sont représentées par une association ou une dépendance de a vers b sur le diagramme de classes. On appelle ces relations des relations directes d'utilisation.

Relation directe d'utilisation. *Il y a une relation directe d'utilisation de la classe a vers la classe b sur un diagramme de classes UML, s'il existe une association ou une dépendance de a à b. Dans le cas d'associations non dirigées, les dépendances existent de a à b et de b à a. L'ensemble des relations d'utilisations directes pour un diagramme de classes est noté SDU. La Figure 55 illustre les deux types de relations entre classes : une association entre les classes BookState et Book et une dépendance entre BookEvent et Book.*

La notion de relation directe peut être étendue à une relation transitive d'utilisation. En effet, une relation peut exister entre deux classes a et b même s'il n'y a ni association ni dépendance entre elles ; ceci est dû aux relations transitives.

Relation transitive d'utilisation. *La fermeture transitive de SDU définit toutes les relations transitives d'utilisation entre les classes du diagramme de classes. On note Θ l'ensemble des relations transitives d'utilisation. Si l'implantation finale permet l'instanciation d'une relation transitive d'utilisation d'un objet o1 de la classe a avec un d'objet o2 de la classe b, on parle alors d'une* relation transitive effective *de a vers b.*

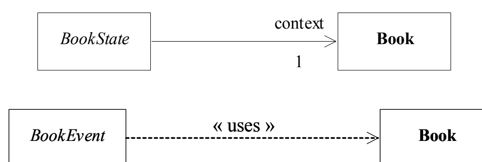


Figure 55 – Relations directes entre classes

Par exemple, la Figure 56 illustre deux relations entre les classes BOOKEVENT et BOOK. Une dépendance stéréotypée « uses » entre les deux classes spécifie une relation directe d'utilisation. La seconde relation est transitive à travers la classe BOOKSTATE. La classe BOOKEVENT dépend de BOOKSTATE qui dépend de BOOK. La classe BOOKEVENT peut donc dépendre de manière transitive de BOOK lorsqu'elle utilise des services de BOOKSTATE. Cette relation est une relation effective si les méthodes de BOOKSTATE appelées par BOOKEVENT utilisent des services fournis par BOOK. Le diagramme de séquence de la Figure 57 illustre une relation transitive effective entre BOOKEVENT et BOOK. Lorsqu'un objet de type BORROW (donc de type BOOKEVENT) appelle la méthode `borrow()` de la classe ORDERED, cette méthode appelle la méthode `setCurrent_state()` de BOOK. Il apparaît donc qu'un objet de type BORROW dépend effectivement d'un objet de type BOOK via un objet de type BOOKSTATE.

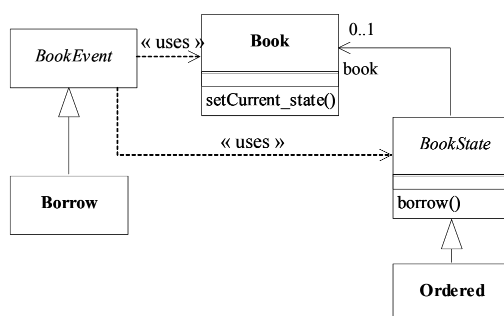


Figure 56 – Relation transitive d'utilisation entre BOOKEVENT et BOOK via BOOKSTATE

Définissons maintenant les notions d'interaction de classes et d'auto-utilisation. Ces interactions sont potentielles puisqu'elles sont détectées à partir du diagramme de classes qui est une vue abstraite du logiciel. Nous définissons alors la notion d'interaction d'objets qui est une interaction effective puisque les relations entre les objets courants sont impliquées. Certaines d'entre elles peuvent être détectées au niveau du modèle grâce aux diagrammes d'objets ou aux diagrammes de séquence. Cependant, ces diagrammes n'offrant qu'une vue partielle du système, et étant susceptibles de changer, ils ne peuvent pas être employés pour détecter toutes les interactions effectives dans le système. Ces deux notions sont formalisées dans les définitions suivantes.

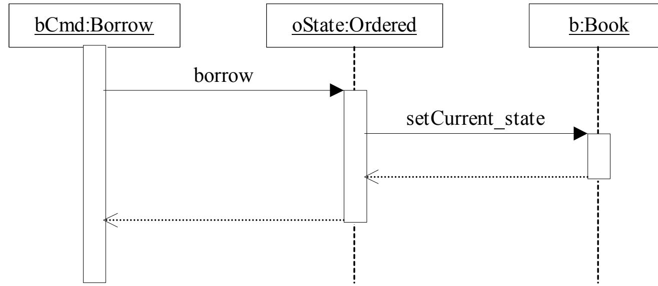


Figure 57 – Diagramme de séquence illustrant une relation transitive effective entre BOOKEVENT et BOOK

Interaction de classes (interaction potentielle). *Une interaction de la classe A vers la classe B se produit ssi :* $\exists R_i \in \Theta$ *et* $R_j \in \Theta$, $R_i \neq R_j$, *tel que A* R_i *B et A* R_j *B.*

Une interaction d'auto-utilisation se produit autour de la classe A si A R_i *A. La Figure 58 illustre les deux configurations génériques.*

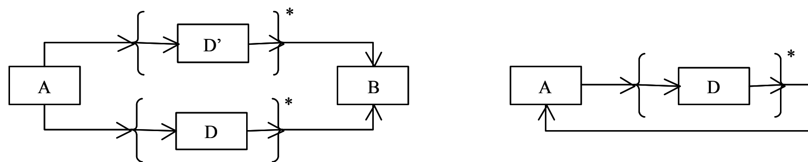


Figure 58 – Interaction de classes et auto-utilisation

La Figure 56 illustre une interaction de classe entre BOOKEVENT et BOOK. Deux dépendances entre ces classes sont impliquées dans l'interaction. Il est à noter que de manière générale, une interaction de classes peut impliquer plus de deux relations transitives d'utilisation.

Interaction d'objets (interaction effective). *Il existe une interaction d'objets d'un objet o1 de la classe A à o2 de la classe B ssi :* $\exists R_i \in \Theta$ *et* $R_j \in \Theta$, $R_i \neq R_j$ *, tel que A* R_i *B et A* R_j *B, ou A* R_i *A.*

> R_i *et* R_j *sont des relations transitives effectives pour* o_1 *and* o_2*. Dans le cas où les relations ne sont pas effectives pour les objets, l'anti-pattern est* infaisable*.*

Par exemple, si le diagramme de séquence de la Figure 57 est associé au diagramme de classes de la Figure 56, l'interaction de classe entre BOOKEVENT et BOOK est une interaction d'objets.

Propriété. *Le nombre d'interactions de classe est une borne supérieure du nombre d'interactions d'objets.*

La propriété est évidente si on fait l'hypothèse que le code est dérivé (probablement automatiquement à l'aide d'un AGL approprié) du modèle.

Maintenant que nous avons défini les interactions de classe et d'objet, nous pouvons donner notre critère de test.

Critère de test. *Pour chaque anti-pattern de testabilité :*

- *soit un cas de test est produit qui exécute une interaction d'objets correspondante,*
- *soit un rapport est produit qui justifie que cet anti-pattern est infaisable.*

La génération des cas de test et rapports de test est impossible si le nombre d'interactions de classe est trop élevé. Le but principal de cette étude est la limitation de ces interactions en améliorant le diagramme de classes. En effet, le diagramme doit être aussi proche que possible de l'implantation. Même avec le code, les dépendances effectives ne peuvent pas être statiquement déduites, car les langages OO permettent le polymorphisme et la liaison tardive. Puisque le nombre d'interactions de classes est une limite supérieure du nombre d'interactions d'objets, nous recommandons d'ajouter des informations sur le diagramme de classes. Ces informations devraient permettre de réduire le nombre d'interactions de classes, et donc le nombre d'interactions effectives. Les informations supplémentaires sont des contraintes (par exemple exprimées en utilisant des stéréotypes UML) que le programmeur doit respecter au moment de l'implantation et qui peuvent être statiquement vérifiées. L'emploi de la vérification statique au niveau du code permet alors de réduire l'effort de test. Par exemple, si le stéréotype «instantiate» est employé pour une dépendance de A vers B, le code de la classe A devrait appeler seulement les méthodes de création de B. Cette contrainte de conception peut être vérifiée statiquement sur le code.

5.3.4 Exemple pour la génération de test

Cette section illustre la génération de cas de test satisfaisant le critère de test défini précédemment sur l'exemple du système de gestion de livres. Deux interactions d'auto-utilisation et une interaction de classes sont présentes dans le diagramme de la Figure 53:

- AU1 de BOOK vers elle-même à travers BOOKEVENT
- AU2 de BOOK vers elle-même à travers BOOKSTATE
- CI entre BOOKEVENT et BOOK par deux chemins différents (un direct et un autre passant à travers BOOKSTATE).

Le critère de test précise qu'il faut produire un cas de test pour chaque interaction. Si les interactions ne sont que potentielles, un rapport établissant l'absence d'interaction effective doit être établi. Un cas de test consiste donc à créer une instance de Book et à appeler des méthodes sur cet objet.

a Test d'AU1

Un cas de test couvrant AU1 doit appeler une méthode de BOOK qui utilise l'ensemble commands. De plus, cette méthode doit appeler une méthode de BOOKEVENT qui utilise BOOK. Dans la classe BOOK, seule la méthode `manageEvent()` utilise l'ensemble commands. Dans toutes les classes concrètes implantant les évènements les méthodes sont de la forme suivante :

```
execute(Book b){...}
```

Donc toutes les classes d'évènements utilisent la classe BOOK. Comme `manageEvent()` utilise un objet de type BOOKEVENT, alors un cas de test qui appelle `manageEvent()` couvre l'interaction AU1. Voici un exemple d'un tel cas de test (TC1) :

```
public void testManageEvent(){
  Book b = new Book();
  b.manageEvent("setDamaged");
}
```

b Test d'AU2

Un cas couvrant AU2 devrait appeler une méthode dans la classe BOOK qui utilise l'attribut `currentState`. En fait, il n'y a pas de méthode dans la classe BOOK qui appelle une méthode sur cet attribut. L'interaction d'auto-utilisation AU2 est donc infaisable.

c Test de CI

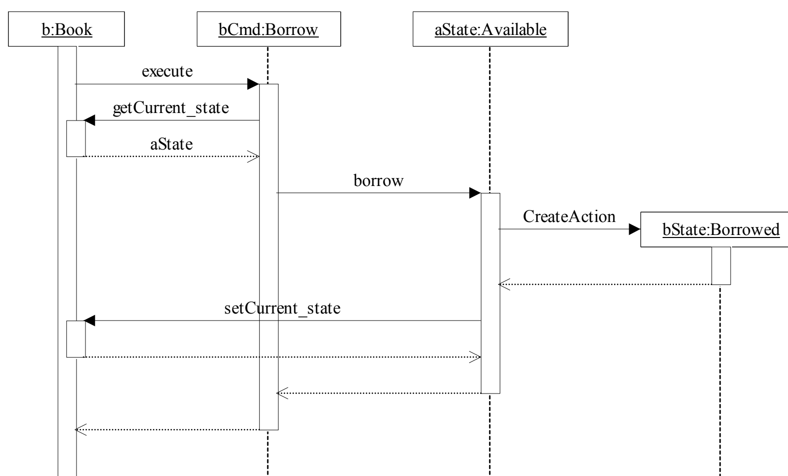


Figure 59 – Diagramme de séquence pour le cas de test n°3

Le cas de test doit nécessairement appeler une méthode de BOOK qui utilise l'ensemble commands, pour atteindre la classe BOOKEVENT, qui est la source de l'interaction CI. Puis il faut couvrir le chemin direct de BOOKEVENT à Book et le chemin de BOOKEVENT à BOOK qui traverse BOOKSTATE. En écrivant le cas de test TC1, nous avons vu qu'un appel à la méthode `manageEvent()`couvre la relation de Book à BOOKEVENT et aussi la relation directe de BOOKEVENT à BOOK. Le chemin direct de BOOKEVENT à BOOK est donc couvert par un cas de test qui appelle la méthode `manageEvent()`de BOOK.

Pour couvrir le second chemin de BOOKEVENT à BOOK (à travers BOOKSTATE) l'appel à `manageEvent()` doit couvrir la relation entre BOOKEVENT et BOOKSTATE. Pour cela, il faut appeler un événement dont le traitement dépend de l'état du livre, dans ce cas la méthode `execute()` de l'événement est de la forme suivante :

```
execute(Book b){b.getState();...}
```

Ensuite, si une transition du diagramme d'états est exécutée par l'appel d'évènement, alors la relation entre BOOKSTATE et BOOK est couverte à son tour, puisque dans ce cas la méthode dans l'état concret appelle une méthode sur l'attribut `context`. Par exemple, la méthode `borrow()` dans la classe AVAILABLE doit changer l'état du contexte auquel il est associé, puisque l'événement d'emprunt dans l'état `available` déclenche une transition vers l'état `borrowed`. Voici le code correspondant :

```
class Available{
  public void borrow(){context.changeState(new Borrowed());}
}
```

Pour résumer le troisième objectif de test, le cas de test TC2 suivant couvre l'interaction CI entre BOOKEVENT et BOOK, et la Figure 59 donne le diagramme de séquence représentant l'exécution de ce cas de test.

```
public void testManageEvent(){
  Book b = new Book(); //the book is in the ordered state
  b.manageEvent("deliver"); //puts the book in the available state
  b.manageEvent("borrow");
}
```

Le Tableau 7 résume les résultats du test de l'implantation du système de gestion de livre (Figure 53) à partir du critère de test de la section précédente. Ce tableau présente le statut de chaque anti-pattern présent dans le diagramme (faisable ou non), puis quels sont les anti-patterns couverts par chaque cas de test. Il y a en fait beaucoup plus de trois anti-patterns qui devraient être testés à cause des sous-classes de BOOKSTATE et BOOKEVENT. La section 5.4.3 détaille le calcul de la complexité de ces anti-patterns. Cette complexité correspond au nombre maximum d'anti-patterns qui peuvent apparaître et qui doivent être testés.

Tableau 7 – Rapport de test pour le gestionnaire de livres

	SU1	SU2	CI
Statut		infaisable	
TC1	X		
TC2			X

L'outil JTracor, que nous avons évoqué au chapitre précédent, peut être utilisé pour aider la génération de cas de test satisfaisant le critère de test. En effet, comme l'outil produit des traces d'exécution pour des programmes écrits en Java, il est possible de connaître les interactions effectives entre objets et quelles méthodes ont été appelées sur ces objets. Les traces obtenues avec JTracor pour l'exécution de TC1 et TC3 sont données dans l'Annexe B.

5.4 Modélisation des anti-patterns

Dans les sections précédentes, nous avons exprimé les propriétés qu'un modèle abstrait du diagramme de classes devrait vérifier pour pouvoir indiquer exactement toutes les interactions de classe pour un système. Le modèle que nous proposons est fondé sur un graphe dont nous donnons ici les règles de dérivation à partir d'un diagramme de classes UML. Un tel graphe s'appelle un Graphe des Dépendances de Classes (GDC). Après avoir défini précisément la construction de ce graphe et quelles informations il comporte, nous donnons des règles sur la topologie du graphe qui déterminent formellement les anti-patterns. Le GDC sert de base pour appliquer des algorithmes de graphe classiques pour détecter les anti-patterns et mesurer leur complexité.

5.4.1 Construction d'un graphe à partir d'un diagramme de classes d'UML

Un graphe de dépendances de classes peut être construit automatiquement à partir d'un diagramme de classes. Les définitions suivantes précisent quelles informations le GDC doit contenir, et comment les obtenir à partir d'un diagramme de classes. L'ensemble de toutes les classes d'un système est désigné par $\mathcal{C}$, et M(c) désigne l'ensemble des méthodes d'une classe $c \in \mathcal{C}$.

Un Graphe De Dépendance De Classes (GDC). *Un graphe de dépendance de classe est une paire GDC=(X, Γ), où :*

- *X est l'ensemble des sommets, chaque sommet représentant une classe d'un système orienté objet. Une classe est représentée par un simple sommet.*
- *Γ est l'ensemble de paires $(x,y) \in X^2$, appelé ensemble des arcs orientés $((x,y) \neq (y,x))$ Un arc entre deux sommets, x et y, représente une dépendance de la classe représentée par x à la classe représentée par y. Un arc est marqué par le type de dépendance qui existe entre les deux classes, à savoir dépendances d'utilisation (association ou dépendance UML) ou bien héritage.*

Un GDC peut être facilement construit à partir d'un diagramme de classes UML, les règles de transformation sont résumées Figure 60. Ces transformations sont rendues explicites dans les définitions suivantes. Notez que, puisqu'il y a un sommet pour chaque classe et chaque sommet représente une et une seule classe, dans les définitions suivantes, le sommet correspondant à une classe c s'appelle simplement c.

Étiquettes d'arc. *Chaque arc dans un GDC représente une dépendance entre deux classes d'un système orienté objet. L'arc entre les sommets $c \in C$ et $d \in C$ est marqué par le type de dépendance existant entre c et d. Les dépendances peuvent être de deux types : utilisation (étiquette U) si c utilise d ou héritage (étiquette I) si $c \neq d$ et c hérite de d.*

Étiquette U. *Nous associons un ensemble de méthodes à l'étiquette U qui correspond à l'ensemble de méthodes dans M(d) employé par la classe c. La valeur par défaut de cet ensemble de méthodes est M(d) (tant que nous ne connaissons pas le sous-ensemble M(d) utilisé par c). Cette transformation est illustrée dans la Figure 60(a). Dans le cas d'une dépendance d'utilisation stéréotypé par «instantiate» ou «create» entre les classes c et d, l'ensemble de méthodes associées à l'étiquette U est (created ()) et indique que c appelle seulement la méthode de création de classe d à travers cette relation d'utilisation.*

Étiquette I. *L'étiquette d'héritage est dérivée en deux étiquettes : I-child et I-parent (Figure 60(b)). Si $c \in C$, $d \in C-\{c\}$, et c hérite directement de d, alors il y a un arc (d,c) marqué I-child et un arc (c,d) marqué I-parent. Si d est une interface pure, l'arc(c,d) n'existe pas (dans ce cas, c ne peut pas employer des méthodes de d, Figure 60(c)), et si la classe d dépend d'autres classes, les arcs marqués U de d n'existent plus, mais sont déplacés aux enfants concrets de d (la Figure 60(d)).*

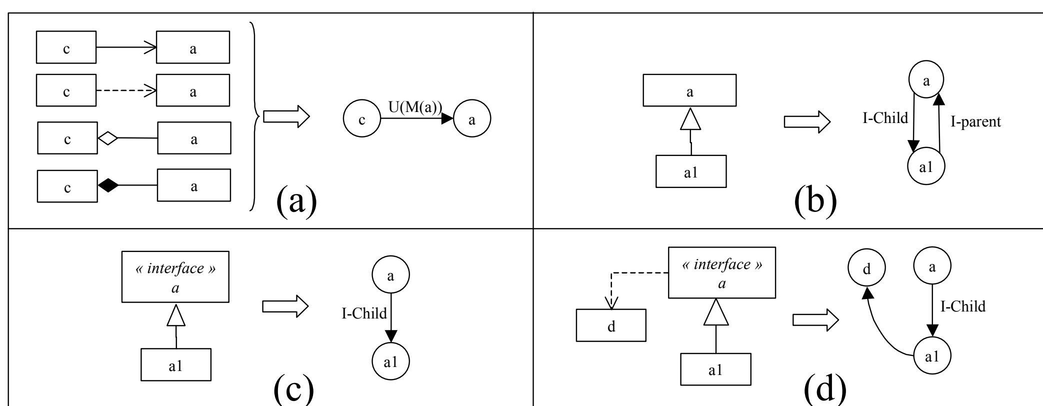


Figure 60 – Les transformations de base d'un diagramme de classes UML en un GDC

Du point de vue du test, nous avons besoin d'une dépendance du parent à l'enfant, parce que toute référence à un objet de type parent peut désigner un objet fils. Ainsi, toute relation à une classe parent se propage vers chaque classe fille. La dépendance de la classe fille à la classe parent est évidente: c emploie d quand elle appelle une méthode m héritée de d.

La Figure 61 donne l'exemple d'un graphe de dépendance de classes obtenu à partir d'un petit diagramme de classes, en appliquant les règles de transformation données dans les définitions ci-dessus.

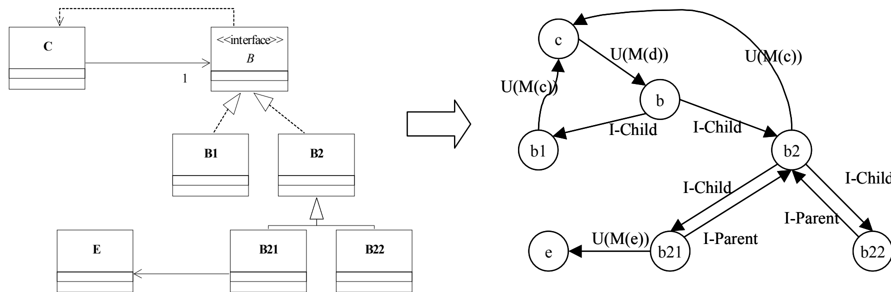


Figure 61 – Exemple GDC

5.4.2 Détecter des anti-patterns à partir du GDC

Nous revenons maintenant à la notion d'anti-pattern, et les définissons en termes de topologie remarquable dans un graphe. Pour cela nous rappelons les définitions de chemin et de cycle dans un graphe qui seront nécessaires pour définir les anti-patterns.

Chemin. *Un chemin C dans un GDC est une séquence de sommets* $C = [x_i1, x_i2, x_i3, \ldots, x_ik]$ *tels que :*

- $(x_i1, x_i2) \in \Gamma$, $(x_i2, x_i3) \in \Gamma, \ldots, (x_ik\text{-}1, x_ik) \in \Gamma$
- x_i1 *est l'origine du chemin et est appelé origine(C)*
- x_ik *la fin, fin(C)*
- *les* x_ij $(2 \leq j \leq k\text{-}1)$*, sont les sommets intermédiaires (cet ensemble de sommets est appelé sommetsIt(P)).*

Cycle. *Soit C un chemin, C est un cycle ssi fin(P) = origine(P).*

Chemin et cycle élémentaire. *Un chemin élémentaire est une séquence de sommets tels qu'il n'y ait jamais deux fois le même sommet. Un cycle élémentaire est un chemin élémentaire dans lequel seul le sommet d'origine est répété une fois.*

Sur la Figure 61, [c, b, b2, b22] ou [c, b, b2, b21, e] sont des chemins élémentaires, mais [c, b, b2, b21, b2] n'en est pas un. De même, [c, b, b2, c] est un cycle élémentaire, mais pas [c, b, b2, b21, b2, c].

Interaction De Classe (IC). *Il existe une interaction de classe IC(c,d) de la classe* $c \in \mathcal{C}$ *à la classe* $d \in \mathcal{C}\text{-}\{c\}$*, s'il existe au moins deux chemins élémentaires* P_1 *et* P_2 *tels que* $P_1 \neq P_2$ *et* $(\text{origine}(P_1) = \text{origine}(P_2)=c) \wedge (\text{end}(P_1) = \text{end}(P_2) = d) \wedge (\text{sommetsIt}(P_1) \neq \text{sommetsIt}(P_2))$.

Un chemin traversant une hiérarchie d'héritage ne doit la traverser que dans une seule direction, c.-à-d. qu'on ne doit avoir que des arcs allant des sommets enfants aux sommets parents ou des arcs allant des sommets parents aux sommets enfants.

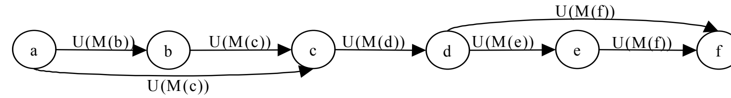


Figure 62 – IC sur un GDC

Cette définition de l'interaction ne tient compte que des interactions unitaires. Par exemple, sur le GDC de la Figure 62, deux interactions sont détectées : IC(a, c) et IC(d, f), tandis que la plus grande interaction IC(a, f) n'est pas détectée. Nous supposons qu'il est suffisant de ne détecter que les interactions, puisque la résolution de IC(a, c) et IC(d, f) résout également IC(a, f).

Auto Utilisation (AU). *Il existe une interaction d'auto-utilisation AU(c) sur la classe c ∈ C, s'il existe un cycle élémentaire d'origine c. L'anti-pattern concerne tous les nœuds du cycle. Un cycle traversant une hiérarchie d'héritage doit croiser la hiérarchie seulement dans une direction, c.-à-d. qu'il ne doit y avoir que des arcs allant des sommets enfants aux sommets parents ou des arcs allant des sommets parents aux sommets enfants.*

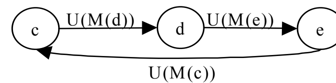


Figure 63 – AU Sur un GDC

La Figure 63 montre un graphe sur lequel une interaction AU(c) peut être détectée: il y a un cycle élémentaire du sommet c au sommet c. Tout comme l'interaction IC, la définition de l'interaction AU donnée ci-dessus considère seulement les interactions unitaires.

5.4.3 Complexité des anti-patterns

La complexité d'un anti-pattern peut maintenant être formalisée en tenant compte du polymorphisme dans le système. La complexité d'une interaction augmente quand un ou plusieurs chemins impliqués traverse un graphe correspondant à une hiérarchie d'héritage.

Complexité d'une interaction. *Soit* $\{P_1 \ldots P_{nbPaths}\}$ *un ensemble de chemins correspondant à une interaction de classe IC, la complexité de l'interaction est liée à la complexité des différents chemins de la façon suivante:*

$$\text{complexité}(CI) = \sum_{i=1}^{nbPaths} \left(\text{complexité}(P_i) \bullet \sum_{j>i} \text{complexité}(P_j)\right)$$

Chemin de descendance. *Dans une hiérarchie d'héritage, un chemin de descendance est l'ensemble des classes croisées par un chemin allant de la classe racine à une classe feuille.*

Comme défini plus tôt, les chemins impliqués dans une interaction peuvent passer par une hiérarchie d'héritage seulement dans une direction. Nous identifions alors un sous-chemin correspondant à une tranche de la hiérarchie d'héritage allant d'une classe racine à une classe feuille comme le montre la Figure 64. Ce sous-chemin s'appelle un chemin de descendants dans une hiérarchie d'héritage. Si un chemin impliqué dans une interaction passe par une ou plusieurs classes d'un chemin de descendants dans le graphe, la complexité de l'interaction augmente de la façon suivante: s'il y a n classes dans un chemin de descendants, dont aucune n'est une interface pure, la complexité du sous composant est n*(n-1). Chaque classe a une relation avec chacune des (n-1) autres classes et n*(n-1) interactions peuvent se produire et doivent être examinées.

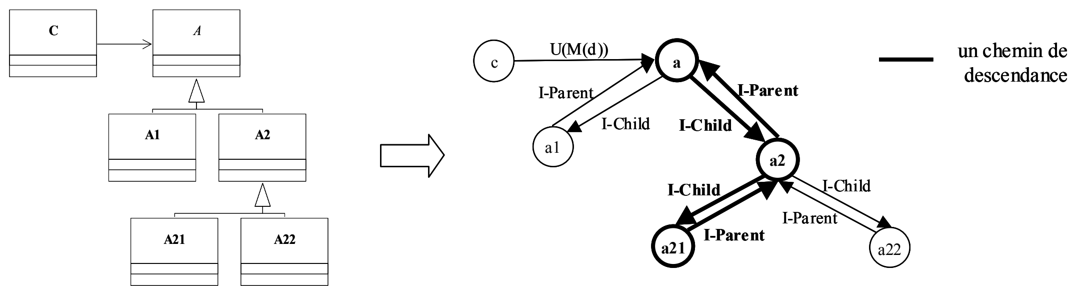


Figure 64 – Tranche dans un SCC correspondant à une hiérarchie d'héritage

Si deux hiérarchies d'héritage sont présentes dans l'interaction, chaque classe d'une hiérarchie peut avoir une relation avec chaque classe de l'autre hiérarchie. Aussi, la complexité d'un chemin est-elle le produit de la complexité associée à chaque hiérarchie croisée par l'interaction.

Complexité d'un chemin dans une interaction de classe. *Soit P un chemin impliqué dans une interaction de classe, et $IH_i, \ldots, IH_{nbCrossed}$ les hiérarchies d'héritage croisées par P, la complexité de P est calculée de la manière suivante :*

$$complexity(P) = \prod_{i=1}^{nbCrossed} complexity(IH_i, P)$$

Les arcs des hiérarchies d'héritage ne sont pas pris en compte dans le calcul de la complexité.

Par la suite, nous définissons la complexité d'un chemin traversant une hiérarchie d'héritage. Plusieurs chemins de descendants dans une hiérarchie d'héritage peuvent augmenter la complexité d'un chemin. Si un chemin passe par une classe qui n'est pas une

feuille dans la hiérarchie d'héritage, il peut y avoir plusieurs chemins de descendants incluant cette classe. Par exemple, sur la Figure 65, le chemin [bEvt, bSt, book] traverse la classe racine de la hiérarchie d'héritage de BOOKSTATE. Puisque BOOKSTATE n'est pas une classe feuille dans l'arbre d'héritage, tous les chemins de descendants commençant par le nœud bSt doivent être pris en compte dans le calcul de la complexité du chemin. Donc, les 5 chemins de descendants [bSt, Or], [bSt, Bd], [bSt, IL, BFx], [bSt, IL, Av], [bSt, IL, Rd] sont impliqués dans le calcul de la complexité du chemin [bEvt, bSt, book].

Complexité d'un chemin passant par une hiérarchie d'héritage. *Soit IH une hiérarchie d'héritage et P un chemin croisant IH, la complexité d'IH pour P est l'addition de la complexité de* $dp_1, \ldots, dp_{nbDP}$ *, des chemins de descendance dans IH influençant la complexité de P.*

$$complexiéy(IH,P) = \sum_{i=1}^{nbDP} \text{complexité}(dp_i)$$

La complexité d'un chemin de descendants correspond au nombre maximum de dépendances entre les classes dans ce chemin. Dans le pire cas, chaque classe dépend de toutes les autres, ainsi, s'il y a n classes dans le chemin, il y aura donc tout au plus, n*(n-1) interactions.

Complexité d'un chemin de descendants. *Soit DP un chemin de descendance de hauteur h, la complexité pour DP est alors :*

$$complexité(dp) = h \bullet (h - 1)$$

Dans la suite, ces définitions sont illustrées par le calcul de la complexité sur l'exemple du système de gestion de livres.

5.4.4 Mesure de la complexité des anti-patterns : système de gestion de livres

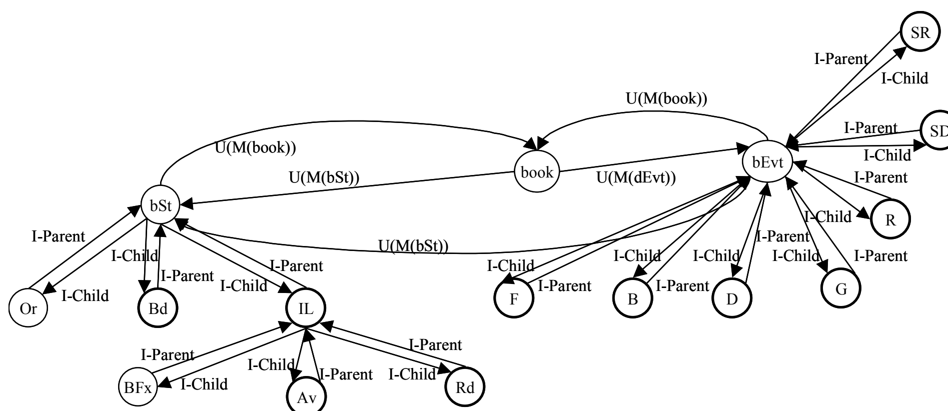


Figure 65 – GDC pour le gestionnaire de livres

Le GDC pour le gestionnaire de livres de la Figure 53 est donné Figure 65. A partir de ce graphe, nous détaillons maintenant le calcul de la complexité de l'interaction de classes entre BOOKEVENT et BOOK. La complexité de cette interaction est le produit de la complexité des deux chemins impliqués : C1 = [bEvt, book] et C2 = [bEvt, bSt, book].

Même si C1 se résume à un arc simple entre deux nœuds, il a une complexité non nulle puisque la classe BookEvent fait partie d'un arbre d'héritage. La complexité de C1 est donc la somme des complexités de chaque chemin de descendants impliqué. Comme la classe BOOKEVENT est la racine de l'arbre d'héritage, tous les chemins de descendants sont impliqués dans le calcul de la complexité. Tous ces chemins sont de taille 2, et ont donc tous la même complexité. Il y a 7 chemins de descendants, et la complexité pour cet arbre d'héritage est : 7*2*(2-1)=14. C'est aussi la complexité de C1.

Le chemin C2 traverse les nœuds bEvt et bSt qui correspondent à deux classes racines d'arbres d'héritage. La complexité de l'arbre sous le nœud bEvt est 14. La complexité pour le second arbre d'héritage est l'addition de tous les chemins de descendance dans cet arbre. Deux chemins sont de longueur 2 et trois de longueur 3, la complexité est : 2*(2-1)+2*(2-1)+3*(3-1)+3*(3-1)+3*(3-1) = 22. La complexité de C2 est le produit des deux complexités, ce qui correspond au fait que les classes d'un arbre d'héritage peuvent potentiellement interagir avec toutes les classes de l'autre arbre. Le chemin C2 a une complexité de 22*14=308.

La complexité totale pour l'interaction de classe est égale au produit des complexités des chemins C1 et C2. Cette complexité correspond au nombre maximum d'interactions classe à classe, et ne tient pas compte du fait qu'un grand nombre de classes ne dépendent jamais l'une de l'autre. Par exemple la classe SETDAMAGED n'interagit avec aucune classe d'état. La complexité d'un anti-pattern est une borne maximum pour le nombre de relations qui devraient être couvertes au moment du test. Dans la section suivante, nous montrons que l'ajout de stéréotypes sur les associations et dépendances du diagramme de classes permettrait d'ignorer certains arcs au moment du calcul de la complexité d'un anti-pattern, et donc d'obtenir une valeur plus proche de la complexité réelle des interactions d'objets.

5.5 Amélioration de la testabilité du modèle

Améliorer la testabilité du logiciel, en ce qui concerne notre critère de test, signifie éviter les interactions d'objets. Comme nous l'avons suggéré dans la section 5.3.3 une solution consiste à clarifier le modèle, de sorte que le code soit aussi proche que possible de ce que le concepteur veut.

Une première solution consiste à utiliser des *refactorings* [Opdyke"92] sur le modèle pour remplacer, quand c'est possible, une classe abstraite par une classe d'interface. Cette action permet de diminuer la complexité des anti-patterns, puisque dans ce cas les arcs qui entraient dans cette classe disparaissent.

Une seconde solution est fondée sur l'utilisation de stéréotypes UML. En effet, UML permet à un utilisateur de définir des stéréotypes pour associer une sémantique aux éléments d'un modèle. Nous définissons ainsi plusieurs stéréotypes qui indiquent la sémantique des liens impliqués dans des anti-patterns (association, dépendance, agrégation, composition). Grâce à ces éléments de spécification supplémentaires, le programmeur devrait éviter d'implanter une interaction d'objets. Les stéréotypes présentés ici sont analogues d'une certaine manière aux critères de test flots de données [Rapps"85], qui identifient la "définition" et l'"utilisation" des variables dans un programme. Ce modèle de test classique vise à déterminer le flot de données, la "ligne de vie" des variables au niveau unitaire.

Nous proposons quatre stéréotypes:

- « create » : c'est un stéréotype standard d'UML. Ce stéréotype sur un lien d'une classe A à une classe B signifie que les objets du type A appellent la méthode de création (ou constructeur) de la classe B. Si aucun stéréotype « use » n'est attaché au même lien, seule la méthode de création peut être appelée.
- « use » : un stéréotype « use » sur un lien de la classe A à la classe B signifie que les objets du type A peuvent appeler n'importe quelle méthode à l'exclusion des méthodes de création des objets du type B. Les stéréotypes suivants permettent d'affiner cette définition:
    - « use_consult »: une spécialisation du stéréotype « use » où les méthodes appelées ne modifient jamais les attributs des objets de type B.
    - « use_def »: une spécialisation du stéréotype « use » où au moins une des méthodes appelées modifie des attributs des objets de type B.

Par défaut, l'absence de stéréotype sur un lien équivaut à une combinaison de « use » et « create ».

Les stéréotypes sont pris en compte au moment de la génération du graphe en associant une autre valeur aux étiquettes de U. Ceci permet également au concepteur d'estimer l'amélioration du modèle après avoir ajouté des stéréotypes. Cette analyse correspond à l'étape 2 de la méthodologie proposée dans la section 5.2.2. L'utilisation des stéréotypes modifie l'identification des interactions d'objets comme suit.

Assertion 1. Interactions d'objets. *Soit P1 et P2 deux chemins de la classe c vers la classe d, définissant une interaction de classe entre c et d. Soit e1 l'arc entrant du nœud fin(P1), et e2 l'arc entrant le nœud fin(P2), une interaction d'objet existe ssi :*

e1 et e2 sont stéréotypés « use » ou « use_def ».

Assertion 2. Interaction d'objets d'auto utilisation. *Soit P un chemin de la classe c à elle-même, définissant ainsi une interaction d'auto utilisation pour c. Soit e l'arc entrant du nœud fin(P), une interaction d'objet d'auto utilisation existe ssi :*

e est stéréotypé « use » ou « use_def ».

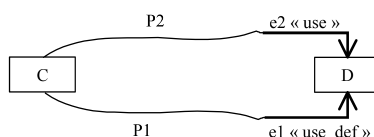


Figure 66 – Une interaction de classe entre C et D

La Figure 66 illustre une interaction de classes. Les chemins allant de C à D et qui ne se terminent pas avec un arc stéréotypé « use » ou « use_def », ne causent pas une utilisation contradictoire de la classe D par la classe C. Chaque chemin peut être examiné indépendamment en utilisant les critères de test qu'Alexander propose dans [Alexander"00].

La conformité du code avec les contraintes des stéréotypes peut être vérifiée automatiquement. Par exemple, la vérification d'un stéréotype «use-consult» de C à D consiste à vérifier que C n'appelle que les méthodes de consultation de D et que les méthodes de consultation de D ne modifient jamais l'état de D. Dans le cas d'une rétro-ingénierie, ces stéréotypes peuvent être déduits d'une analyse statique en utilisant, par exemple, la méthode décrite par Richner et Ducasse dans [Richner"99].

La section suivante illustre des problèmes potentiels de testabilité sur trois exemples, et illustre différentes solutions pour éviter des problèmes réels au niveau de l'implantation. Des stéréotypes sont présentés directement par le concepteur qui veut spécifier le logiciel avec plus de précision. Par ailleurs, nous voyons dans la suite que des stéréotypes peuvent également être présentés automatiquement, au méta-niveau d'UML pour des design patterns. Cette méta programmation d'un modèle employant des collaborations paramétrées est détaillée dans le section 5.8.3, après la discussion de la testabilité des design patterns.

5.6 Exemples d'application

Cette section illustre les différentes techniques pour améliorer la testabilité d'un diagramme de classes sur trois exemples : le système de gestion de livres, un serveur de réunion virtuelle et un compilateur. L'étude de ces résultats permet de repérer les points dans une conception qui peuvent entraîner une implantation difficile à tester s'ils sont mal interprétés.

5.6.1 Le gestionnaire de livres

Le GDC pour le système de gestion de livres est présenté Figure 65. Dans la section 5.4.4 nous avons calculé la complexité pour l'interaction de classe IC(BOOKEVENT, BOOK) et nous avons mentionné que de nombreuses interactions sont infaisables, et ne devraient donc

pas être prises en compte pour le calcul de la complexité. Par la suite, plusieurs stéréotypes ont été définis dans le but de préciser le rôle des liens dans le diagramme de classes. Ceci permet de prendre en compte de manière différente les différents types de relations au moment du calcul de la complexité.

L'interaction de classe IC(bEvt, book) peut être supprimée en spécifiant que la dépendance entre les classes BOOKEVENT et BOOK n'est utilisée qu'en lecture. La dépendance peut donc être stéréotypée «uses_consult», et il n'y a plus d'interaction de classes. Cependant, le chemin reliant BOOKEVENT à BOOK en traversant BOOKSTATE reste toujours complexe pour le test. Cette partie du diagramme peut être améliorée pour la testabilité en transformant les classes BOOKEVENT et BOOKSTATE en classes d'interface. Ceci éviterait les interactions entre ces classes et leurs classes filles et ferait passer la complexité de ce chemin de 308 à 56. De la même manière, la complexité des deux interactions d'auto-utilisation SU1(book) et SU2(book) peut être réduite en transformant les classes BOOKEVENT et BOOKSTATE en classes d'interface.

5.6.2 Serveur De Réunion Virtuelle

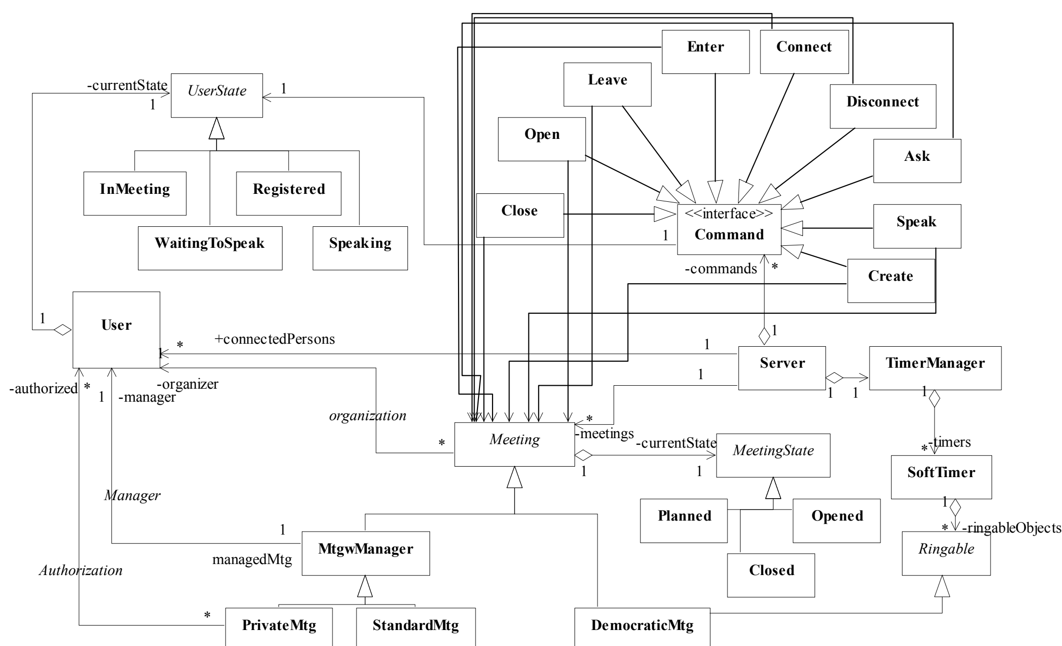


Figure 67 – Le serveur de la réunion virtuelle

Le serveur de réunion virtuelle vise à simuler des réunions de travail. Une fois relié au serveur, un participant peut entrer ou sortir d'une réunion, parler, ou planifier de nouvelles réunions. Il existe trois types de réunions :

– Réunions standards où le participant qui a la parole est désigné par un modérateur (nommé par l'organisateur de la réunion).

- Réunions démocratiques : des réunions standards où le modérateur est un robot FIFO (le premier participant à demander la permission de parler est le premier à parler).
- Réunions privées : des réunions standards avec l'accès limité à un ensemble défini de participants.

Toutes les commandes possibles sont réifiées et héritent de l'interface command. Les états internes possibles d'un client et d'une réunion sont contrôlés par le pattern state. La Figure 67 donne le diagramme de classe du serveur de la réunion virtuelle.

Le graphe de dépendance de classe du serveur de la réunion virtuelle est donné Figure 68. Un grand nombre d'interactions de classe sont détectées sur ce modèle, et nous ne les détaillons pas toutes ici. Nous insistons juste sur les configurations intéressantes et montrons que même sur un modèle simple (29 classes), beaucoup de problèmes de test émergent.

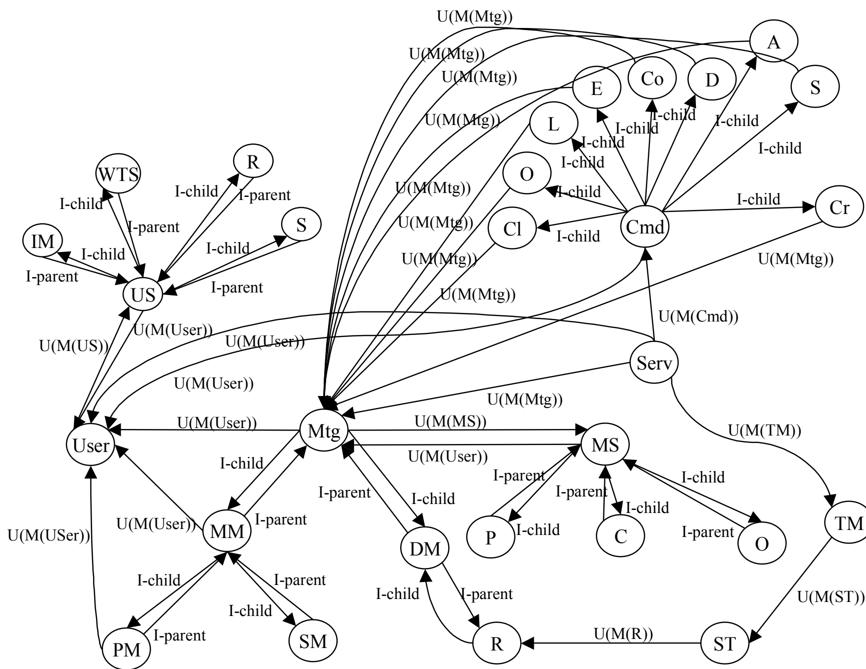


Figure 68 – GDC du serveur de la réunion virtuelle

Il y a deux interactions d'auto-utilisation autour des noeuds User et Mtg , qui sont dues à l'utilisation du pattern state. Néanmoins, si le pattern state est correctement implanté, cette configuration ne produira en fait aucune interaction d'objets, puisque l'interface state emploie seulement la méthode de son contexte pour modifier l'état courant du contexte, tandis que le contexte délègue tout traitement dépendant de l'état à son état courant.

Une interaction de classes est détectée entre Serv et R : le serveur peut accéder à l'interface RINGABLE par l'intermédiaire de la classe TIMERMANAGER ou par la classe DEMOCRATICMTG. Cette interaction de classes pourrait être facilement évitée en transformant la classe abstraite MEETING en interface. Une configuration intéressante

d'interactions de classes imbriquées existe entre User, Serv, Mtg et MM. Il y a une interaction de classes IC(Serv, User) entre Serv et User d'un coté, et IC(Mtg, User) entre Mtg et User de l'autre. Notez que IC(Serv, User) inclut IC(Mtg, User).

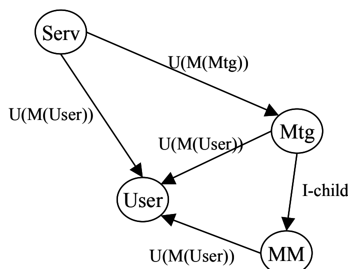


Figure 69 – Configuration des interactions de classes incluses

Deux remarques peuvent être faites sur cette configuration particulière isolée sur la Figure 69. D'abord, suivant la manière dont l'interaction imbriquée IC(Mtg, User) sera résolue, l'interaction de classes IC(Serv, User) n'est pas nécessairement résolue. Deuxièmement, même s'il n'est pas possible de supprimer CI(Mtg, user), CI(Serv, user) peut être résolue, par exemple en ajoutant au modèle un stéréotype «use-consult» sur l'association de server à meeting. De cette configuration, nous pouvons déduire que les interactions de classes peuvent être combinées de différentes manières; dans certains cas, toutes les interactions de classes ne sont pas nécessairement prises en compte (comme sur la Figure 62), dans d'autres cas, il est nécessaire de traiter toutes les interactions de classes (comme sur la Figure 69).

D'autres interactions de classes peuvent être détectées, par exemple de Mtg à User (où Mtg peut accéder à user directement, par MM ou par le MM et PM) ou de Serv à Mtg.

5.6.3 Une architecture de compilateur

La Figure 70 donne une architecture orientée objet pour un compilateur pris dans [Jézéquel"99]. Cette architecture comprend une classe SCANNER qui produit des jetons, une classe PARSER qui produit un arbre syntaxique abstrait en utilisant NODE_BUILDER, et PROGRAM_NODE représentant un nœud abstrait dans l'arbre syntaxique.

Le graphe de dépendance de classe dérivé de cette architecture est donné sur la Figure 71. Deux interactions de classe peuvent être détectées sur ce graphe. La première, IC(Fa, PN), est due aux deux chemins [Fa, NB, PN] et [Fa, NV, PN]. La deuxième interaction potentielle, IC(NV, PN), est due aux chemins [NV, Fa, NB, PN] et [NV, PN]. Les deux interactions semblent simples puisqu'elles ne concernent que quatre classes, liées par des relations simples d'utilisations. Cependant, leur complexité croît énormément en raison des onze classes dans la hiérarchie d'héritage de PROGRAM_NODE : neuf chemins de descendance de taille trois sont impliqués dans IC(Fa,PN) et IC(NV,PN). La complexité globale de cette hiérarchie est

$\sum_{i=1}^{9}(3 \cdot (3 - 1)) = 54$ La hiérarchie d'héritage NODE_VISITOR a un plus petit impact sur la complexité puisqu'elle a seulement deux classes. La complexité pour cette hiérarchie est 4.

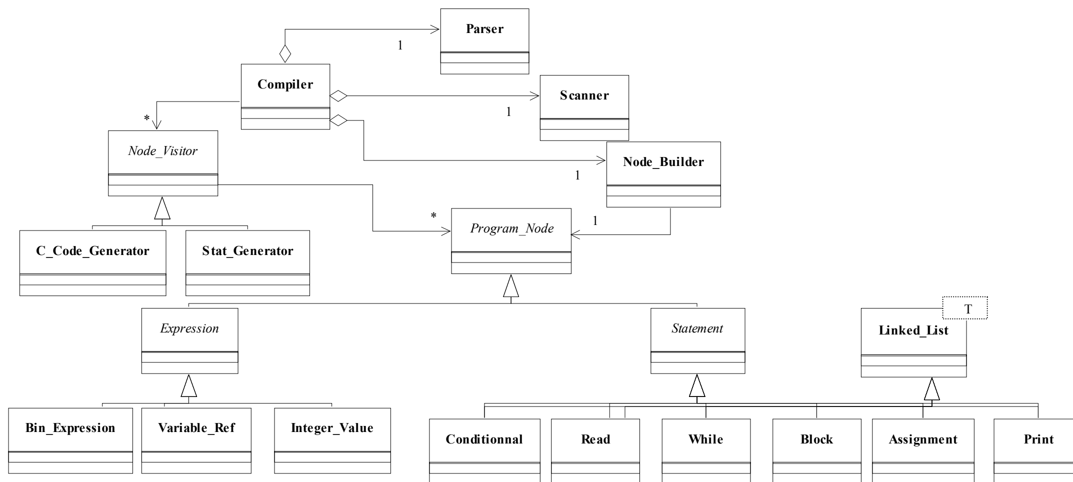


Figure 70 – Architecture Du Compilateur

Comme tous les chemins concernés dans les interactions croisent les mêmes hiérarchies, ils ont tous la même complexité: 54×4 = 216. De la même manière, les deux interactions ont la même complexité qui est le produit de la complexité des deux chemins : 216×54=11664.

Ici, le modèle peut être raffiné avec des stéréotypes sur des associations de façade à NODE_VISITOR et de façade à NODE_BUILDER. En effet, les instances de façade devraient employer des instances de NODE_VISITOR seulement pour des consultations, l'association est ainsi stéréotypée « use_consult ». L'association de FAÇADE à NODE_BUILDER devrait être stéréotypée « use_def » puisque les instances de façade pourraient changer l'état des instances de NODE_BUILDER. Si ces stéréotypes sont sur le modèle, le programmeur ne devrait pas implanter d'interactions d'objet.

Jusqu'ici nous nous sommes concentrés sur des anti-patterns de testabilité sur le modèle globale d'un système. Nous avons montré comment détecter ces configurations sur un diagramme de classes, puis nous avons proposé des règles pour préciser le modèle afin de réduire la complexité de tels anti-patterns. De toutes nos études, il s'est avéré qu'il peut être très difficile de résoudre ces problèmes de testabilité de manière globale pour un système. Une solution serait de résoudre les problèmes dans des sous-systèmes. Les sections 5.7 et 5.8 se concentrent sur les design patterns, qui apparaissent en tant que sous-systèmes logiques et bien définis dans un diagramme de classes global. La section 5.7 détaille les avantages en termes de testabilité de concevoir en utilisant des patterns et la section 5.8 se concentre sur la résolution des problèmes de testabilité pour des applications de design pattern.

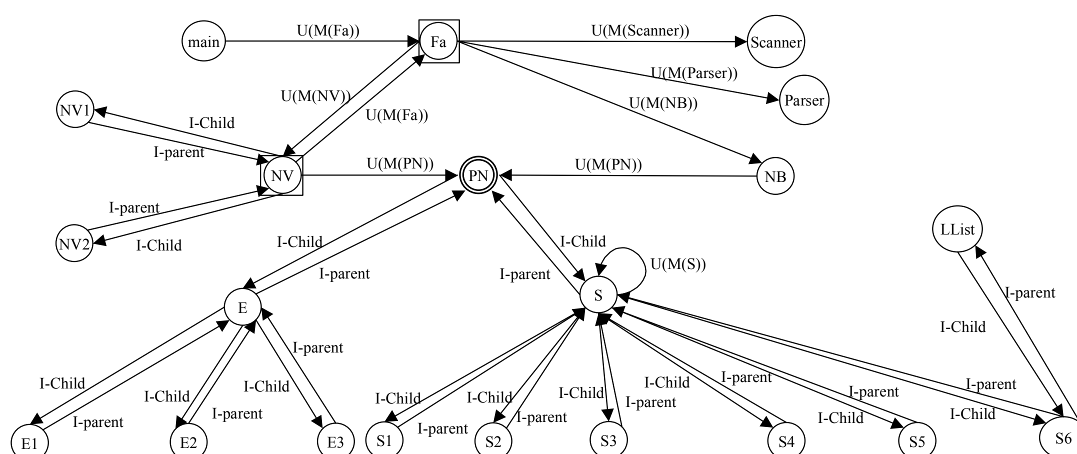


Figure 71 – GDC pour l'architecture du compilateur

5.7 Design Patterns pour la testabilité du modèle

Jusqu'a récemment, une architecture OO finale apparaissait souvent comme un ensemble complexe de classes agissant l'une sur l'autre en l'absence de sous-ensembles logiques émergeant du modèle global. Cependant, grâce à UML, des méthodologies systématiques telles que CatalysisSM [D'Souza"98] offrent maintenant une approche de décomposition de l'architecture. Ces méthodologies aident à concevoir un logiciel orienté objet comme une succession de raffinements, depuis les premières étapes de l'analyse jusqu'à l'exécution. En particulier, les design patterns [Gamma"95] peuvent servir comme base pour une telle amélioration. À partir d'un diagramme de classes d'analyse, les design patterns aident le concepteur en proposant des solutions de conception pour résoudre des problèmes dans un contexte particulier, et ainsi transformer le diagramme en un autre plus orienté vers l'implantation. Les design patterns correspondent alors a des sous-ensembles dans le diagramme de classes, et peuvent être considérés comme une structure intermédiaire entre l'architecture globale et la classe simple. Cette décomposition du système fournit une solution intéressante pour résoudre certains problèmes à un niveau local quand la solution au niveau global est trop complexe.

5.7.1 Concevoir par cristallisation de patterns

Design patterns. *Les design patterns représentent des solutions aux problèmes qui surgissent quand un logiciel est développé dans un contexte particulier. Les design patterns peuvent être considérés en tant que microarchitectures réutilisables qui contribuent à une architecture du système globale; ils saisissent les structures dynamiques et statiques et les collaborations entre les principaux participants du modèle de conception.*

La Figure 72 montre un premier modèle orienté objet (étape d'analyse) pour un client de messagerie instantanée. La Figure 73 illustre une conception détaillée finale possible après

plusieurs étapes d'amélioration, et montre des instances de design patterns dans des ellipses selon la norme UML.

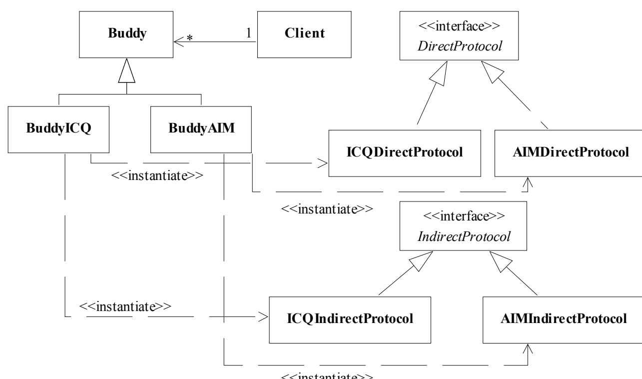


Figure 72 – Un client de messagerie instantanée (diagramme d'analyse)

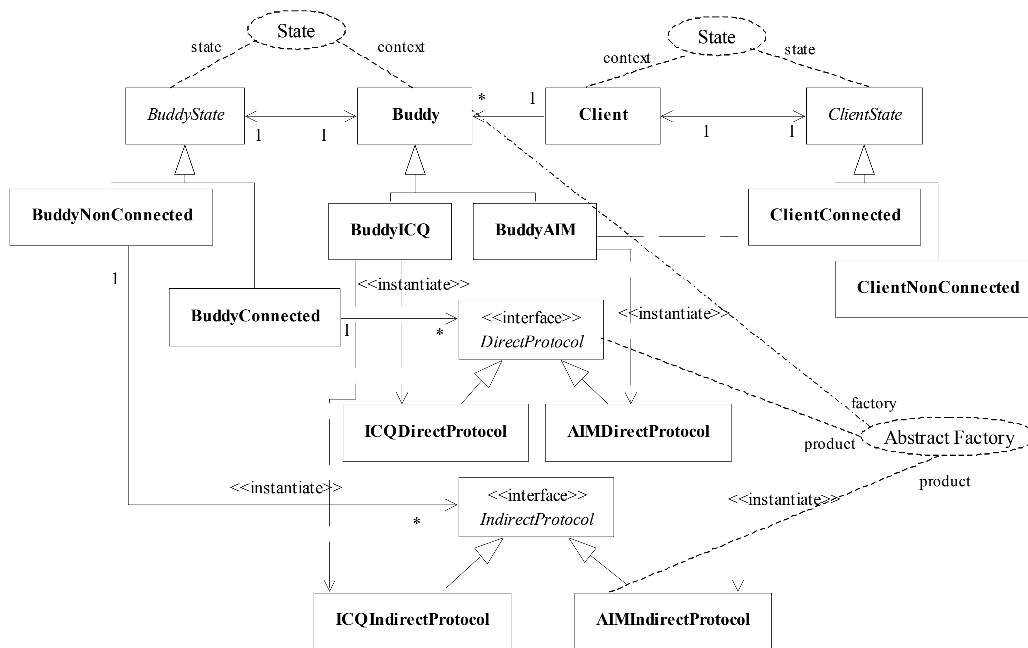


Figure 73 – Un modèle « final » possible pour le client de messagerie instantanée

L'architecture de la Figure 73 est une conception orientée objet typique obtenue après des étapes de cristallisation. Une étape de cristallisation comporte l'adaptation d'un design pattern à un diagramme de classes.

Cette approche est une méthode largement répandue pour passer d'un premier diagramme d'analyse vers un autre plus orienté vers l'exécution. Après l'application d'un design pattern, les classes principales d'analyse demeurent sur le diagramme. De nouvelles classes et liens apparaissent et quelques relations d'associations sont supprimées. Dans l'exemple de la

Figure 73, le diagramme de classes d'analyse a été modifié à travers trois étapes indépendantes qui correspondent à l'adaptation et la combinaison du pattern Abstract Factory et de deux patterns State. Les nouvelles relations et classes introduites pendant le raffinage du modèle en employant des design patterns semblent rendre la conception très dure a tester ; en particulier chaque classe peut potentiellement interagir avec toute autre. Cependant, la méthodologie de développement (cristallisation des design patterns à partir d'une première analyse) nous permet réellement de considérer la totalité du modèle plus comme une composition de microarchitectures que comme un ensemble monolithique de classes inter-connectées. Par conséquent, la complexité globale est décomposée en une combinaison de complexités de microarchitectures, et la tâche de test est ainsi simplifiée. En effet, une fois que les sous-ensembles sont identifiés dans le modèle, plusieurs problèmes de test peuvent être résolus à un niveau local (microarchitecture). Les questions sont:

- Quelles sont les améliorations pour la testabilité obtenues en utilisant les design patterns?
- Les problèmes de testabilité peuvent-ils être localisés (confinés) au sous-ensemble du diagramme qui correspond à l'application d'un design pattern?

Nous répondons à la première question dans le contexte d'une comparaison de testabilité entre l'utilisation isolée du design pattern State et l'implantation procédurale classique d'une machine d'états en utilisant une seule classe.

Nous discutons de la deuxième question - définition de règles/contraintes sur le modèle qui évitent des anti-patterns de testabilité - à travers une solution qui attache des contraintes à un modèle telles que l'implantation soit testable.

5.7.2 Application du design pattern State pour améliorer la testabilité

Nous distinguons la programmation orientée objet de la programmation procédurale par l'effort consacré à la modélisation. Nous considérons le logiciel orienté objet comme un logiciel pour lequel la majeure partie de l'effort est consacrée à la conception de l'architecture, la décomposition modulaire (classe), et le couplage entre les classes (en utilisant autant que possible des constructions orientées objet particulières telles que l'héritage et le polymorphisme). Nous illustrons les différences entre programmation fonctionnelle et OO grâce à deux implantations d'une machine d'états.

L'exemple, pris dans [Jézéquel"99], implante la machine d'états liée à la classe mailer (Figure 74). Cette classe définit une interface publique pour envoyer et recevoir des messages indépendamment du statut de la connexion réseau. Quand la connexion réseau est disponible, les messages sont simplement expédiés. Si elle est indisponible, les messages à diffuser sont stockés dans le serveur de messagerie en attendant la transmission quand la connexion réseau sera restaurée.

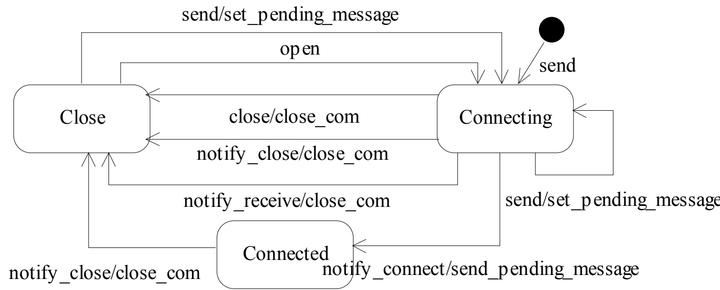


Figure 74 – Machine d'états pour la classe Mailer

Une implantation procédurale de cette machine d'états consiste à coder la machine comme une seule classe. Par exemple, le corps de chacune des méthodes dépendantes de l'état courant est une grande conditionnelle. De telles méthodes (par exemple `send()`) vérifient la valeur de l'état courant, et exécutent un traitement différent en fonction de cette valeur. Ceci est illustré sur la Figure 75. Il y a au moins deux arguments contre l'emploi de ce type de mise en œuvre. D'abord, il tend à rendre le code peu clair, difficile à lire, à maintenir ou réutiliser. En second lieu, il rend le logiciel difficile à étendre: dans ce cas particulier, l'ajout d'un nouvel état impose l'ajout d'un nouveau cas dans chacune des méthodes dépendantes de l'état, c'est-à-dire que des changements sont exigés au niveau du code de plusieurs méthodes. Par conséquent, le risque de commettre une erreur augmente.

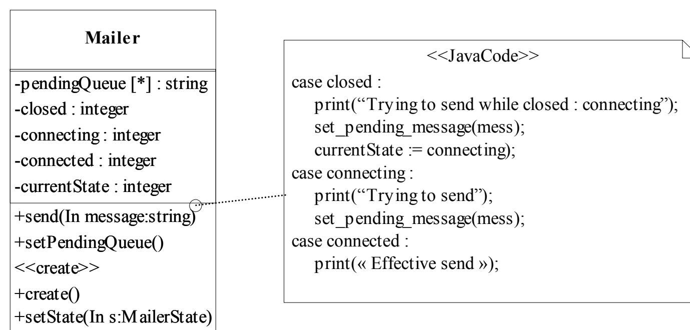


Figure 75 – Machine d'états implantée dans une seule classe

Par ailleurs, cette implantation doit être testée pour valider chaque méthode. Une fois que l'instance de la classe est dans un état particulier, chaque méthode est examinée pour cet état et pour chaque transition existant depuis cet état. Ainsi, le nombre d'instructions pour examiner cette classe est de l'ordre du nombre moyen d'instructions requis pour mettre la classe dans un état donné, multiplié par le nombre moyen de transitions sortantes, plus le nombre d'instructions dont on a besoin pour tester son comportement.

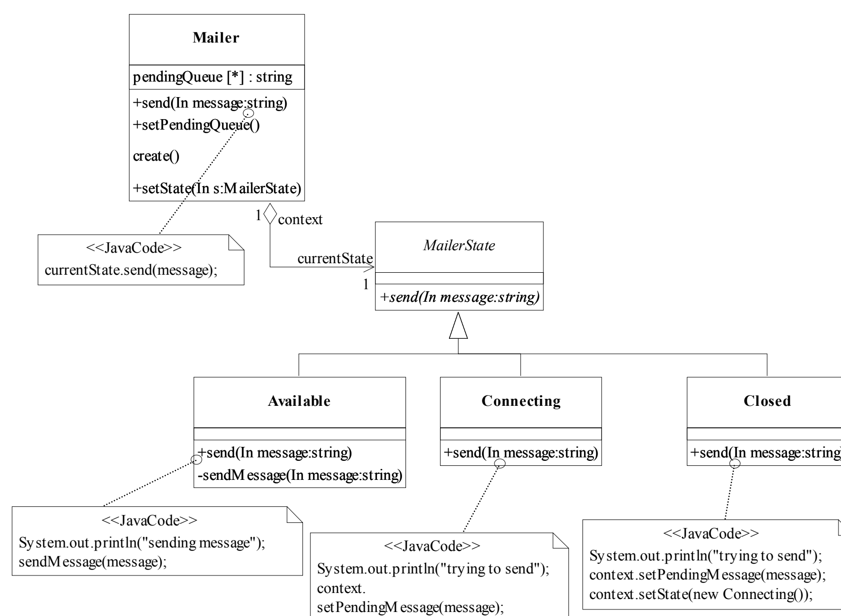


Figure 76 – Machine d'états implantée avec le design pattern state

Maintenant, implantons la machine d'états avec le pattern state [Gamma"95], l'architecture est donnée Figure 76. Elle contient la classe MAILER et l'interface MAILERSTATE. Le MAILERSTATE rassemble chaque méthode dont le traitement dépend de l'état courant. Il y a une classe concrète d'état pour chaque état possible. Dans la classe MAILER, le traitement des méthodes dépendantes de l'état courant est délégué à l'objet currentState, qui peut être lié dynamiquement à n'importe quel classe héritant de la classe MAILERSTATE. Par conséquent, le contrôle de la valeur de l'état ne se fait plus explicitement dans la classe mailer. Le bon traitement d'une méthode est fait grâce à la liaison dynamique. Ceci rend la classe MAILER beaucoup plus claire, et complètement indépendante de sa machine d'états. Les deux problèmes détectés dans la solution précédente sont résolus. Le code de chaque méthode est maintenant beaucoup plus petit (chaque méthode exécute seulement une opération), et ainsi plus lisible. De plus, l'architecture facilite le changement de la machine d'états. Par exemple, ajouter un nouvel état consiste à ajouter une nouvelle classe concrète pour cet état.

Plusieurs éléments rendent la conception "pattern" plus testable que la conception "classique", même si elle semble plus compliquée en termes de nombre de classes ou en termes de couplage entre ces classes. D'abord, la structure de contrôle complexe de l'implémentation précédente a disparu. Le contrôle n'a pas besoin d'être testé en tant que tel, puisqu'il est implanté grâce à la liaison dynamique. Ensuite, le test est simplifié puisque les comportements spécifiques pour chaque état sont isolés dans différentes classes. Ainsi, il n’y a plus besoin du préambule qui met l'objet dans un état particulier avant de le tester. En effet,

l’exécution d'une méthode pour un état donné se fait directement en exécutant cette méthode sur une instance de la classe correspondant à cet état.

Cependant, de nouveaux problèmes de test surgissent, qui semblent spécifiquement liés à la conception orientée objet, et plus particulièrement, à la microarchitecture correspondant à l’application du design pattern.

5.8 Analyse de testabilité des design patterns

Nous nous intéressons maintenant à la résolution des anti-patterns de testabilité à un niveau local, pour diminuer la complexité de la testabilité du système global (cf. section 5.4.3). Nous illustrons l'analyse de testabilité proposée (voir section 5.4) sur deux microarchitectures, correspondant aux applications des design patterns Abstract Factory et State. Le choix d'utilisation des diagrammes UML/OMT pour décrire la structure du design pattern dans [Gamma"95] mène à plusieurs erreurs d'interprétation. Par ailleurs, l'introduction de stéréotypes appropriés pour clarifier la conception doit être faite à la main. Nous proposons ainsi d'employer des diagrammes de collaboration comme représentation pour décrire les design patterns [Le Guennec"00; Sunyé"00]. D’une part, cette représentation est plus adaptée à la description des aspects statiques et dynamiques des patterns. D’autre part, elle peut servir à une application automatique des design patterns sur un diagramme de classes. La définition des contraintes pour la testabilité au niveau méta permet alors une application automatique des stéréotypes sur le modèle.

5.8.1 Analyse de testabilité de State

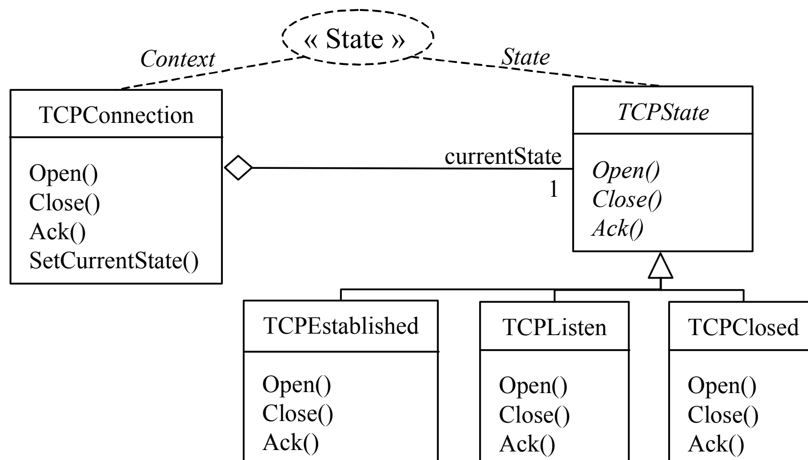


Figure 77 – Une application du Design Pattern State

Nous appliquons l’analyse de testabilité (section 5.4) sur une instance du design pattern State dont le diagramme de classes est donné sur la Figure 77. Ceci nous permet de détecter avec plus de précision les problèmes de testabilité spécifiques à l'application du pattern State évoqués à la fin de la section 5.7. Dans cette application particulière l'association entre TCPCONNECTION et TCPSTATE est bidirectionnelle : TCPCONNECTION garde une

référence sur son état courant, et chacun des états concrets doit connaître son contexte pour changer la valeur de l'état courant. Cette association bidirectionnelle cause des interactions multiples d'auto utilisation autour de TCPCONNECTION.

Une solution pour améliorer la testabilité du diagramme de classes de la Figure 77 est d'employer autant que possible des interfaces à la place des classes abstraites ; ceci évite les liens partant des classes filles vers les classes mères. Si seules les classes feuilles dans la hiérarchie d'héritage (classes qui n'ont aucun descendant) sont des classes concrètes, la complexité de l'interaction traversant cette hiérarchie est réduite (il n'y a plus d'interactions entre les classes dans la hiérarchie).

Le test de l'application du pattern State consiste à vérifier que les méthodes de la classe TCPCONNECTION dépendant de l'état courant ne font pas autre chose qu'une délégation vers l'attribut `currentState`. Par ailleurs, il faut vérifier si les états concrets n'appellent pas les méthodes du contexte qui sont déléguées à l'attribut `currentState`. Si ces contraintes sont vérifiées, il n'y a aucune interaction puisque nous pouvons assurer qu'il n'y aura pas d'interactions d'auto utilisation.

En appliquant le pattern State comme dans [Gamma"95], trois anti-patterns de testabilité sont détectés (des interactions potentielles d'AU). Si toutes les classes abstraites sont converties en interfaces, les trois interactions demeurent mais sont moins complexes. Enfin, si la contrainte de délégation du modèle est vérifiée, il n'y aura plus d'interaction dangereuse.

5.8.2 Analyse de testabilité d'Abstract Factory

La Figure 78 montre une application du design pattern Abstract Factory prise de [Gamma"95]. Une interaction de classes apparaît entre la classe CLIENT et chaque classe produit concret (WINDOW ou SCROLLBAR concrètes) : la classe Client peut utiliser un produit concret directement ou à travers la hiérarchie d'héritage de WIDGETFACTORY.

Comme pour le pattern State, une règle informelle pour améliorer la testabilité du diagramme de classes de la Figure 78 consiste à employer autant d'interfaces que possible. Pour tester l'application du pattern Abstract Factory, nous devons vérifier si la délégation de CLIENT à WIDGETFACTORY crée tous les objets et ne fait pas autre chose. Si l'application du design pattern est correcte, la classe Client doit employer les méthodes de création des produits concrets a travers la hiérarchie d'héritage de WIDGETFACTORY et les autres méthodes par appel direct aux instances des produits concrets. Dans ce cas il n'y a aucun anti-pattern puisque les chemins concurrents entre le client et les produits sont utilisés pour différents buts. En appliquant l'Abstract Factory telle que décrite dans [Gamma"95] (Figure 78), quatre interactions de classe peuvent être détectées. Si toutes les classes abstraites sont converties en interfaces, les quatre interactions demeurent mais sont moins complexes. Enfin, si la délégation est bien appliquée, toutes les interactions de classe sont résolues.

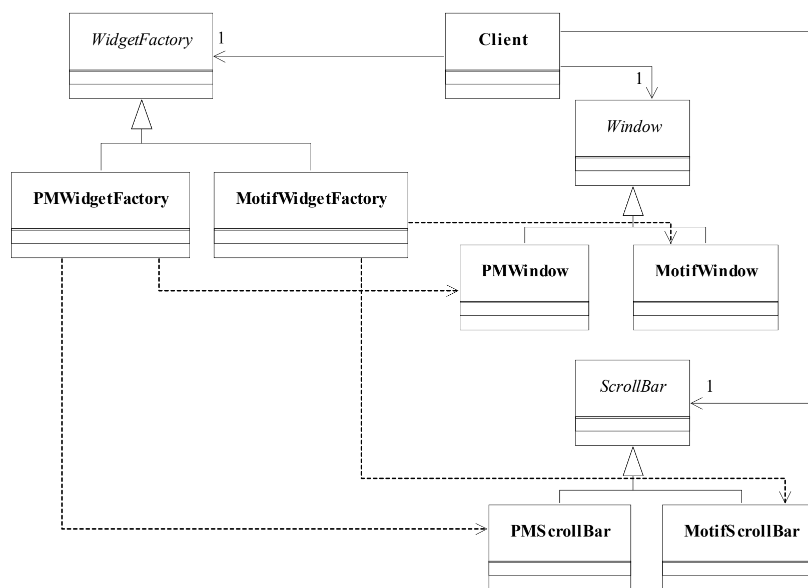


Figure 78 – Une application du Design Pattern Abstract Factory

Pour ce qui concerne la bonne implémentation de la délégation, la solution serait d'exprimer des contraintes d'implantation sur le modèle pour spécifier clairement la délégation. Cependant, de telles contraintes devraient être exprimées au niveau du méta modèle, puisqu'elles concernent des éléments UML et des concepts non modélisés. Plus précisément, il est impossible d'exprimer avec l'OCL qu'une méthode a tout au plus une action à exécuter et que cette action devrait appeler une autre méthode. Un stéréotype «delegation» pourrait être un contournement possible pour ce problème, puisqu'un stéréotype peut avoir des contraintes de méta niveau. Cependant, cette solution n'est pas satisfaisante, puisqu'un stéréotype n'a pas de paramètres et ainsi ne pourra identifier la méthode déléguée. Pour résoudre ce problème, nous employons une autre représentation de design patterns, qui permet d'exprimer des contraintes sur leur implantation, et ainsi de supprimer les anti-patterns.

5.8.3 Définir des contraintes de testabilité au niveau méta

Le diagramme de structure UML/OMT du design pattern dans [Gamma"95] représente des modèles et peut ainsi décrire uniquement une occurrence (ou instance) d'un design pattern. Il ne peut saisir leur vraie structure, exprimée en termes de "rôles", joués par les éléments du modèle (classes, attributs, associations et méthodes).

Nous détaillons ici deux solutions pour exprimer des design patterns au méta niveau, la première est fondée sur des collaborations paramétrés UML, et la seconde consiste à étendre le méta modèle UML. L'idée est que des contraintes de testabilité peuvent être attachées à ce méta niveau de sorte que les stéréotypes appropriés soient automatiquement ajoutés chaque

fois qu'un concepteur crée une instance d'un pattern. Nous illustrons cette approche sur le design pattern Factory

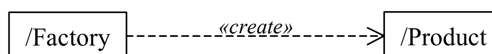


Figure 79 – Collaboration paramétrée

L'utilisation des collaborations paramétrées dans UML est une approche prometteuse pour décrire la structure d'un pattern. Un exemple de cette notation pour modéliser l'Abstract Factory pattern est présenté sur la Figure 79. Ce pattern est composé de deux rôles classificateurs (Factory et Product) et une contrainte de relation «create». Cependant, d'après [Sunyé"00], les collaborations présentent toujours un manque et ne peuvent représenter avec précision ce pattern. Plus précisément, les concepteurs ne peuvent pas indiquer que ce ne sont pas des rôles de classes individuelles, mais de hiérarchies de classes. Le concepteur ne peut pas non plus indiquer que Factory devrait avoir une méthode "créatrice".

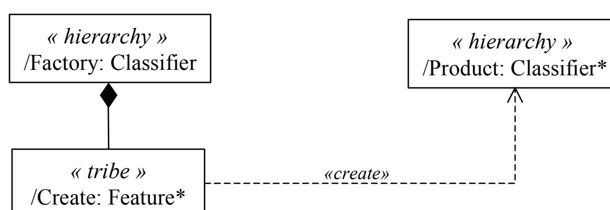


Figure 80 – Le design pattern Abstract Factory

La Figure 80 présente le même pattern, en utilisant une notation différente, proposée dans [Le Guennec"00], où des patterns sont décrits en tant que collaborations du niveau méta, complétées par des contraintes. Ici, ce modèle se compose de trois rôles : Factory, Product and Creator. Le premier rôle est une " hiérarchie", i.e. un ensemble de classes liées par une relation de généralisation. Le second est un ensemble de « hiérarchies ». Finalement, le troisième rôle décrit un ensemble de "tribus". Une "tribu" [Eden"99] est un ensemble d'éléments partageant une signature commune. Chaque élément est possédé par un élément de la hiérarchie Factory. Une contrainte est attachée au rôle creator : il doit effectuer une action simple, l'instanciation d'un product (cette contrainte est concrétisée par le lien «create»). Ceci implique que la multiplicité de Creator est équivalente à celle des hiérarchies de product.

Un exemple de ce modèle est présenté sur la Figure 81. WINDOW et SCROLLBAR jouent le rôle de products, et la hiérarchie de classe WIDGETFACTORY joue le rôle de Factory. Il y a deux clans de méthodes de création dans la hiérarchie WIDGETFACTORY (CreateScrollBar() et CreateWindow()). Cette instance de Factory en combinaison avec une des représentations du méta niveau des Figure 79 ou Figure 80 implique que n'importe quelle dépendance entre WIDGETFACTORY (qui crée une instance du rôle Factory) et WINDOW ou SCROLLBAR (qui crée une instance du rôle Product) est stéréotypée «create». Ainsi, les stéréotypes pour l'amélioration de la testabilité de l'application du pattern sont automatiquement ajoutés pour

éviter l'interaction de classe de la classe client vers les classes product. La tâche suivante est une vérification (qui peut être effectuée statiquement par un CASE tool) de la contrainte «create» sur l'implantation.

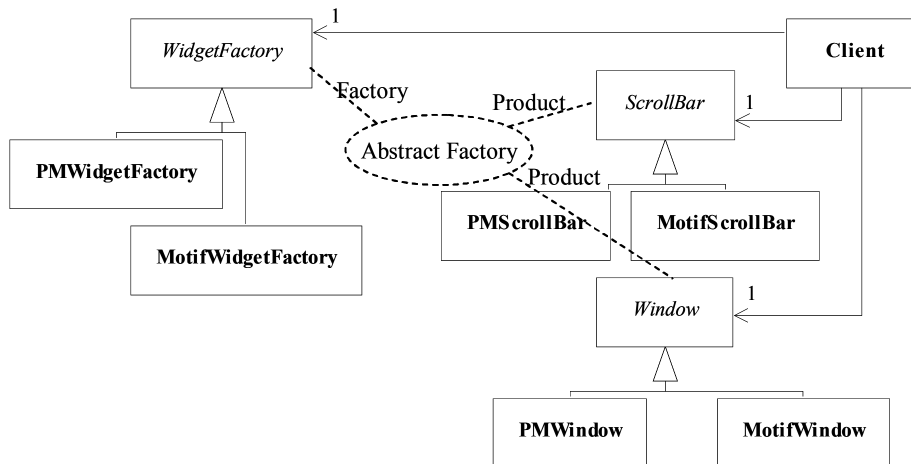


Figure 81 – Un exemple du design pattern Abstract Factory

5.8.4 La grille de testabilité des design patterns

Pour guider les concepteurs, nous proposons la grille suivante qui résume les problèmes de testabilité potentiels pour chaque design pattern lorsque le rôle des associations entre les classes n'est pas défini explicitement. Nous adoptons les instanciations du modèle proposées par Gamma [Gamma"95] puisque nous sommes convaincus que la plupart des concepteurs les prennent comme références. Ceci nous permet de paramétrer chaque pattern comme une fonction des éléments principaux qu'il implique.

Nous nous concentrons sur les design patterns fondamentaux sélectionnés par [Agerbo"98] : Abstract Factory, Bridge, Builder, Composite, Decorator, Flyweight, Proxy, Iterator, Mediator, Memento et State. Il apparaît que les modèles les moins testables sont Mediator et Visitor. Mediator est très semblable au pattern Observer, dont la testabilité a était exhaustivement étudiée par Mc Gregor [McGregor"99]. Quant au pattern Visitor, il est particulièrement difficile à tester à cause d'une utilisation étendue du polymorphisme et une exécution fondée sur le « double renvoi ».

Design Pattern	nombre de participants	nombre d'interactions de classes	nombre d'interactions d'auto-utilisation
Abstract Factory	1 client 1 classe FACTORY abstraite		

	n classes FACTORY concrètes m produits abstraits p produits concrets	p≤ X ≤n*p	aucun
Bridge	***les parametres n'ont pas d'influence***	aucun	aucun
Builder	1 client 1 classe BUILDER abstraite n BUILDER concrets p produits	p≤ X ≤n*p (même type que Abstract Factory)	aucun
Composite	1 client 1 classe composant abstraite n classes composant concrètes	aucun	n
Decorator	1 interface 1 classe composant 1 classe DECORATOR abstraite n classes DECORATOR concrètes	aucun	aucun
Flyweight	Comme abs. fact.	Comme abs. fact.	Comme abs. fact.
Iterator	1 client n CONCRETEAGGREGATE n CONCRETEITERATOR	aucun	2*n (n du client au CONCRETEITERATOR, n du client au CONCRETEAGGREGATE)
Proxy	1 classe SUBJECT abstraite 1 proxy 1 sujet réel	aucun	aucun
Chain of responsibility	1 client 1 classe HANDLER abstraite n classes HANDLER concrètes	aucun	n
Mediator	1 classe MEDIATOR abstraite 1 classe COLLEAGUE abstraite n classes COLLEAGUE concrètes m classes MEDIATOR concrètes	aucun	m*n
Memento	1 MEMENTO 1 ORIGINATOR 1 CARETAKER	aucun	aucun
State	1 classe CONTEXT 1 classe abstraite STATE		

	n classes STATE concrètes (délégation implicite des classes STATE concrètes vers CONTEXT)	no	aucun
Visitor	n éléments visités (CONCRETEELEMENT) p visiteurs (CONCRETEVISITOR)	aucun	n*p

5.9 Conclusion

Au cours de ce chapitre, nous avons étudié la question de la testabilité d'un assemblage de composants. Deux configurations ont été identifiées sur un diagramme de classes, comme pouvant générer des relations entre objets difficiles à tester, et sont appelées des anti-patterns de testabilité. Nous avons donc défini un critère de test pour couvrir ces anti-patterns, et proposé une mesure de complexité associée. Cette mesure, qui est évaluée sur un diagramme de classes, correspond à la testabilité d'un assemblage. En effet, cette mesure permet d'évaluer l'effort nécessaire pour vérifier le critère de test sur un assemblage.

Comme la mesure est évaluée sur une spécification du programme, et pas sur le programme réel, la testabilité obtenue est une valeur prédictive « au pire », i.e., qui sera effective si toutes les relations entre classes détectées deviennent des relations entre objets. Il apparaît alors qu'une façon d'améliorer la testabilité est d'ajouter de l'information sur le diagramme pour se prémunir contre une implantation dure à tester. La seconde contribution de ce chapitre consiste à préciser cette information à l'aide de trois stéréotypes spécifiques qui détaillent le rôle de la dépendance d'un point de vue test (creation only, use consult only ou use definition), dans la logique du test classique de flux de données.

Au cours de la seconde partie de cette étude nous nous sommes concentrés sur les design patterns comme des sous-ensembles logiques dans un diagramme global. En effet, les problèmes de testabilité à un niveau global impliquent souvent un grand nombre de classes et d'interactions et peuvent être ainsi très difficiles à résoudre. Il semblait donc intéressant de trouver un niveau intermédiaire pour améliorer la testabilité. L'analyse de testabilité appliquée sur des instances de design patterns a montré que les problèmes pourraient être résolus à la main ou bien paramétrés grâce aux diagrammes de collaboration (une solution de types transformation de modèles). Comme résultat de l'étude des design patterns, nous avons proposé une grille contenant le nombre d'anti-patterns pouvant apparaître si aucune spécification supplémentaire n'est ajoutée sur les liens d'association du diagramme de classes.

6
Conclusions et perspectives

Au cours de cette thèse, nous nous sommes intéressés à certains mécanismes et constructions particuliers du paradigme objet pour proposer des solutions adaptées au test de ces programmes. Les différentes études décrites dans ce document se sont inscrites dans le cadre d'une méthodologie originale pour la conception de composants logiciels fiables. Cette méthodologie considère le test comme le moyen principal pour la validation d'un programme. Elle impose donc de prendre en compte l'activité de test tout au long des phases d'analyse, de conception et d'implantation. Les travaux présentés dans ce document se sont articulés autour de trois points principaux se rapportant à trois aspects différents du développement d'un composant.

6.1 Contributions majeures

Les trois contributions majeures de cette thèse s'articulent autour de deux axes complémentaires : la conception testable de systèmes orientés objet par assemblage de composants et la validation des composants et assemblages. Les deux sections suivantes résument ces contributions.

6.1.1 Validation de composants

Les chapitres 3 et 4 se sont intéressés à la validation de composants et d'assemblages dans le cadre de la méthode des composants autotestables. Nous avons tout d'abord étudié l'optimisation des cas de test pour un composant, puis la conception par contrat pour la validation d'assemblages de composants.

Au cours du chapitre 3 nous avons proposé une solution pour le problème de la génération et de l'optimisation automatique d'un ensemble de cas de test pour un composant. Nous avons étudié un algorithme génétique pour résoudre ce problème. L'analyse des résultats expérimentaux plutôt décevants nous a conduit à proposer une adaptation de l'algorithme que nous appelons un *algorithme bactériologique*. Le développement d'un outil nous a permis d'effectuer plusieurs expériences qui ont confirmé la pertinence de l'approche pour le test d'un composant.

Le chapitre 4 propose une étude de la conception par contrat pour le test d'un système OO. Nous avons analysé l'apport de cette méthode pour deux aspects du test : la détection et la localisation des erreurs. Pour cela, nous avons défini respectivement, les mesures de robustesse et de diagnosabilité pour des systèmes OO conçus par contrat. Ceci nous a ensuite permis d'estimer l'impact de la densité et de la qualité des contrats sur ces deux mesures. Les résultats ont montré que la simple introduction de contrats réduit rapidement l'effort de détection et de localisation des erreurs, et qu'au-delà d'une densité minimale, la qualité des contrats (leur capacité à détecter un état erroné du système) est le facteur le plus important pour renforcer la robustesse et faciliter le diagnostic.

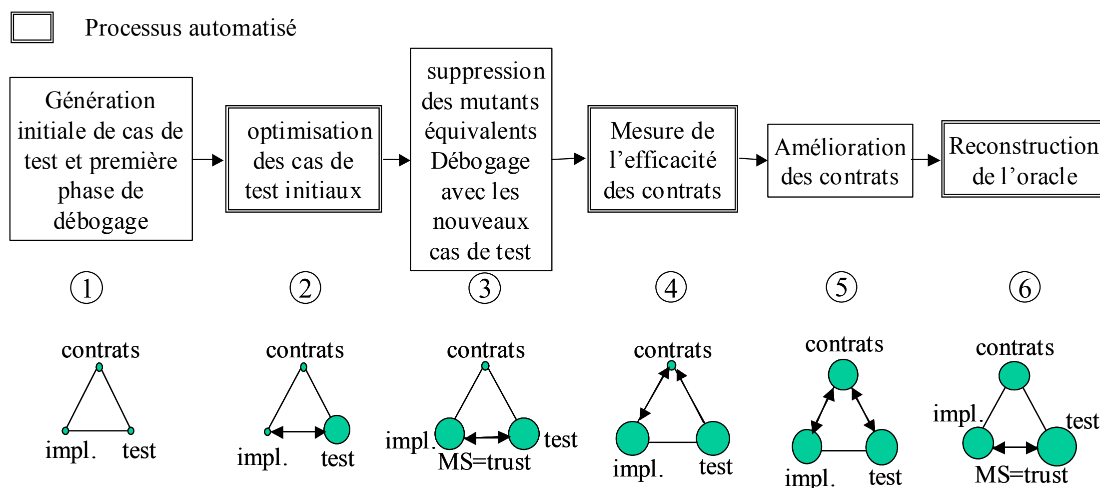


Figure 82 – Processus global pour la conception d'un composant fiable

A partir de ces deux études, nous proposons une approche itérative de la conception de composants autotestables. Cette approche consiste à améliorer les trois facettes du composant autotestable, en six étapes (Figure 82) :

1 une version initiale de l'implantation, des contrats et des cas de test est disponible

2 une analyse de mutation permet d'améliorer les cas de test. Nous avons étudié deux algorithmes évolutionnistes pour automatiser cette étape.

3 les cas de test sont exécutés sur l'implantation. Si des erreurs sont détectées, elles doivent être localisées et corrigées. Dans ce cas, il faut revenir à l'étape 1, puisqu'une modification du programme initial, modifie la génération de mutants, et donc la génération de cas de test.

4 l'efficacité des contrats est mesurée, de nouveau à l'aide d'une analyse de mutation. Cette fois, on n'utilise que les contrats comme oracles pour détecter les mutants. Le score de mutation obtenu correspond à la qualité des contrats.

5 les contrats sont améliorés si nécessaire.

6 l'ensemble des cas de test est minimisé, et exécuté sur le composant. L'état du composant est sauvegardé après l'exécution de chaque cas de test, et c'est ce qui sert d'oracle global.

6.1.2 Testabilité d'un assemblage de composants

La troisième contribution de cette thèse concerne la testabilité des systèmes orientés objet. Cette étude consiste à prendre en compte l'impact des structures de contrôle particulières engendrées au cours d'un assemblage de composants sur la testabilité des systèmes. En effet, l'utilisation de mécanismes tels que la délégation et le polymorphisme entraîne une forte répartition du contrôle à travers tout le système, et implique des interactions complexes entre les objets du système. Nous avons donc défini un critère de test qui vise à couvrir ces interactions. Associée à ce critère, nous proposons une mesure de complexité, calculable à partir d'un diagramme de classes et qui permet d'évaluer la testabilité d'un assemblage dés la conception. Au cours de cette étude de testabilité, nous avons aussi observé le comportement particulier de l'application des design patterns pour le test. Nous avons montré comment, en forçant une utilisation importante de l'héritage et de la délégation, ils suppriment certains problèmes de testabilité des programmes procéduraux, mais introduisent incidemment d'autres difficultés. Or, ces difficultés peuvent être décelées à partir d'un diagramme de classes et peuvent être évaluées avec le critère de test proposé.

6.2 Perspectives

6.2.1 La génération de test pour le diagnostic

Le diagnostic est l'étape du test qui consiste à localiser et corriger une erreur lorsqu'un cas de test a échoué. L'impact de la stratégie de test choisie sur l'efficacité du diagnostic est largement reconnue. Cependant, très peu de travaux se sont intéressés à quantifier cet impact. Sur ce point, nous avons commencé une nouvelle étude qui, à partir de l'analyse des algorithmes de diagnostic, définit des critères de test pour une localisation efficace des erreurs dans un programme. Nous utilisons ensuite un algorithme bactériologique pour générer automatiquement des cas de test qui vérifient ces critères. Les expériences devraient nous permettre d'étudier l'efficacité des cas de

test générés pour la localisation d'erreurs, ainsi que la validité du modèle de diagnosabilité proposé dans cette thèse.

6.2.2 Les contrats comme oracle de test

Au cours de cette thèse, nous avons commencé, en collaboration avec une équipe de l'université de Carleton à Ottawa, à étudier certaines propriétés des contrats qui permettraient d'évaluer leur efficacité pour la détection d'erreur. L'idée initiale était de prioritiser les cas de test qui nécessitent un oracle explicite en présence de contrats. En effet, notre idée était d'observer les traces d'exécution des cas de test, et de classer ensuite les cas de test en fonction de la qualité des contrats le long du flot d'exécution de chacun. On obtiendrait alors un ensemble de cas de test qui exécutent des parties du système dans lesquelles les contrats sont faibles (détectent peu d'erreurs), et pour lesquels ils faudrait écrire un oracle explicite. La définition de l'oracle pour les cas de test exécutant des contrats efficaces pourrait être reportée à plus tard. Faute de temps, ces travaux n'ont pas pu aboutir, mais cela semble être une perspective intéressante pour la diminution de l'effort de test (en facilitant l'écriture de la fonction d'oracle).

6.2.3 Transformation de modèles pour la testabilité

Globalement, et notre travail a tendu vers cela dans un but préventif, il doit être possible d'accompagner toutes les étapes de raffinement de l'analyse et de la conception en transformant les modèles de telle sorte qu'on évite ou limite une dégradation de la testabilité. Un cas particulier de transformation de modèle dont la testabilité peut être contrôlée concerne les design patterns évoqués ci-après. Dans la mesure où – dans le contexte du MDE (Model Driven Enginnering) – on dispose d'un langage de transformation, défini au niveau méta, alors cette technique pourrait être appliquée de façon systématique à toute transformation identifiée et cataloguée. Ainsi, l'ajout de stéréotypes lors de l'application de design patterns serait vue comme une transformation de modèle, et pourrait être automatisée pour rendre les instances de design patterns testables dans un diagramme de classes.

6.2.4 Design patterns et test

En forçant la distribution du contrôle à travers plusieurs classes, les design patterns rendent plus difficiles l'expression de contrats locaux à une classe. Un point intéressant pour la suite serait de reprendre l'étude da la rédaction de contrats efficaces dans le cadre d'une conception fondée sur les patterns, et d'explorer les points suivants :

- existe-t-il des règles spécifiques pour la rédaction de contrats lors de l'application de design patterns ?
- les contrats peuvent-ils être efficaces pour la détection d'erreur lors de l'application de design patterns ?

Annexes

Annexe A Répartition des contrats dans un système OO

Cette annexe présente le détail de la mesure de la répartition des contrats sur cinq systèmes écrits en JAVA. Ces résultats justifient la validité de l'hypothèse de répartition uniforme des contrats nécessaire au modèle de diagnosabilité définit au chapitre 4.

Ces répartitions ont été obtenues en utilisant l'outil JTracor.

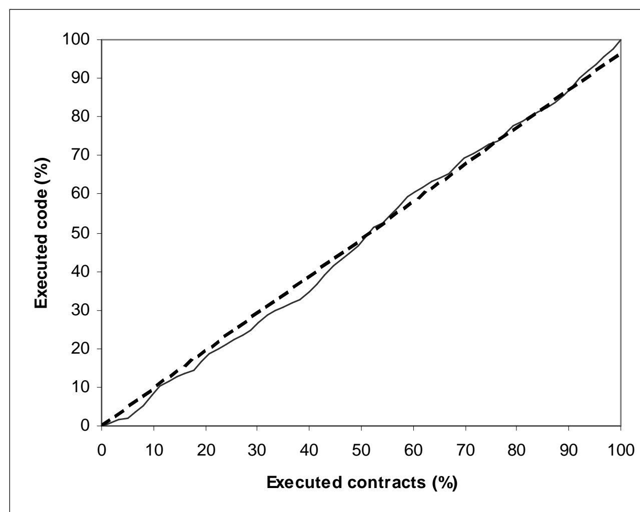


Figure 83 – Répartition des contrats dans une classe List

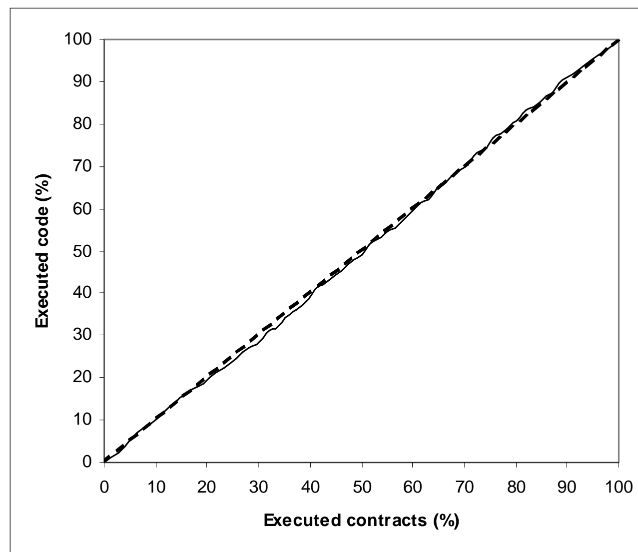


Figure 84 – Répartition des contrats dans un serveur de réunion virtuelle

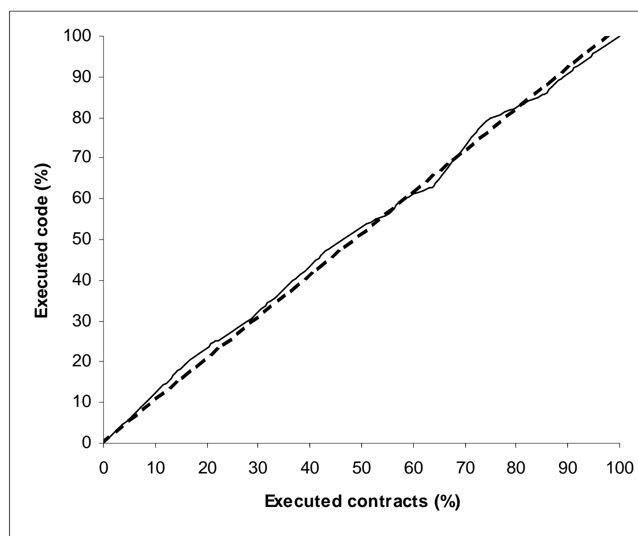


Figure 85 – Répartition des contrats dans la suite de tests de JUnit

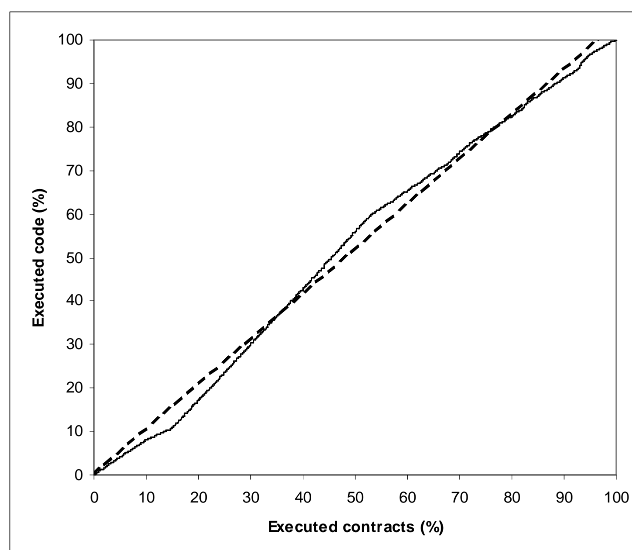


Figure 86 – Répartition des contrats dans le JDK

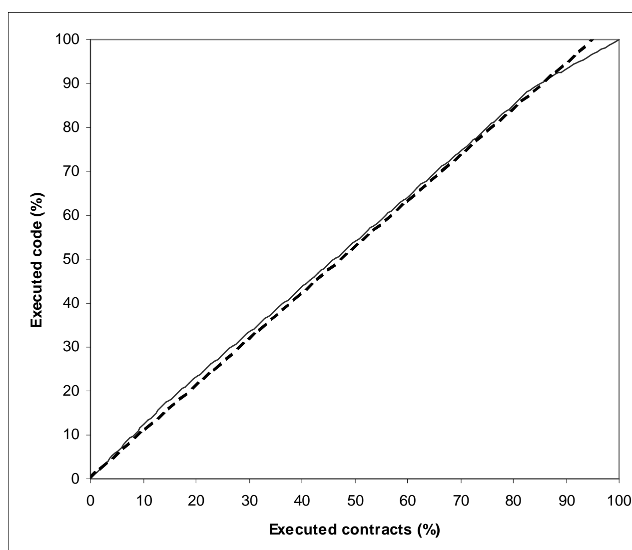


Figure 87 – Répartition des contrats dans Jtree

Annexe B Interactions d'objets pour le gestionnaire de livres

Nous présentons ici les traces d'exécution ainsi que les graphes de relations entre objets (obtenus avec Jtracor) pour les cas de test TC1 et TC3 da la section 5.3.4. JTracor est un framework pour tracer l'exécution de programmes Java, dont nous donnons ici les résultats sous forme textuelle et sous forme de diagrammes d'objets. Cet outil nous a permis d'exhiber deux interactions d'objets, qui rendent l'implantation de ce système peu testable d'après notre critère (section 5.3). Nous commençons par exhiber une interaction d'auto-utilisation effective, puis une interaction d'objets. Enfin, nous donnons les résultats de JTracor sur un autre cas de test qui couvre une interaction plus complexe que les deux premières

AU1

Cette section résume les traces d'exécution pour le cas de test TC1, qui teste l'interaction d'auto-utilisation AU1 de BOOK vers elle-même à travers BOOKEVENT, dans le gestionnaire de livres.

Cas de test TC1 :

```
public void testManageEvent(){
  Book b = new Book();
  b.manageEvent("setDamaged");//the trace is produced from this call
}
```

Trace textuelle pour TC1 :

```
|  |  |  |  |  |->SetDamaged@76.execute
|  |  |  |  |  |  |->Book@82.setDamaged
```

Diagramme d'objets pour TC1:

AU2

Cette section résume les traces d'exécution pour le cas de test TC3, qui teste l'interaction de classes AU1 entre BOOKEVENT et BOOK par deux chemins différents (un direct et un autre passant à travers BOOKSTATE), dans le gestionnaire de livres.

Cas de test :

```
public void testManageEvent(){
  Book b = new Book(); //the book is in the ordered state
  b.manageEvent("deliver"); //puts the book in the available state
  b.manageEvent("borrow"); //the trace is produced from this call
}
```

Trace textuelle pour TC3:

```
|   |   |   |   |   |->Borrow@72.execute
|   |   |   |   |   |   |->Book@82.getCurrent_state
|   |   |   |   |   |   |->Available@81.borrow
|   |   |   |   |   |   |   |->Borrowed@86.<init>
|   |   |   |   |   |   |   |   |->Book_State@86.<init>
|   |   |   |   |   |   |   |->Book@82.setCurrent_state
```

Diagrammes d'objets pour TC3:

IC(BOOKEVENT, BOOK)

Nous détaillons ici un cas de test plus complexe qui couvre l'interaction de classes entre BOOKEVENT et BOOK : un des chemins passe à travers AVAILABLE, RESERVED et BOOKSTATE.

Cas de test :

```
public void testManageEvent(){
  Book b = new Book(); //the book is in the ordered state
  b.manageEvent("reserve"); // the trace is produced from this call
}
```

Trace textuelle :

```
|  |  |  |  |  |->Reserve@66.execute
|  |  |  |  |  |  |->Book@82.getCurrent_state
|  |  |  |  |  |  |->Available@81.reserve
|  |  |  |  |  |  |  |->Reserved@87.<init>
|  |  |  |  |  |  |  |  |->InLibrary@87.<init>
|  |  |  |  |  |  |  |  |  |->Book_State@87.<init>
|  |  |  |  |  |  |  |->Book@82.setCurrent_state
|  |  |  |  |  |  |  |->Book_State@81.reserve
|  |  |  |  |  |  |  |  |->Book@82.inc_nb_res
```

Diagramme d'objets :

:Book@82
execute
getCurrent_state
:Reserve@66
setCurrent_state
reserve
inc_nb_res
:Available@81
<init>
reserve
:Reserved@87
:Book_State@81
<init>
:InLibrary@87
<init>
:Book_State@87

Glossaire

Algorithme évolutionniste : extension d'un algorithme génétique pour inclure les systèmes de classification et la programmation génétique où chaque solution est un programme.

Algorithme génétique : modèle computationnel de la résolution de problème fondé sur les principes de la génétique et les mécanismes de la génétique.

Assertion : prédicat ou combinaison de prédicats.

Cas de test : procédure qui permet d'exécuter le programme sous test avec une donnée de test.

Client : un composant C_i est client d'un composant C_j s'il utilise les services de C_j (i.e. si C_i a un attribut, une variable locale ou un paramètre de méthode de type C_j).

Composant autotestable : composant logiciel dans lequel sont regroupés son implantation, une spécification sous forme de contrats exécutables, et un ensemble de cas de test.

Conception par contrat : méthodologie de conception pour des systèmes orientés objet qui consiste à définir qui consiste à définir les devoirs et obligations des éléments du système. Trois types principaux d'assertions peuvent être définis : des pré et post conditions sont pour les méthodes, et un invariant pour les classes. La précondition définit la condition légale pour utiliser une méthode. La postcondition définit les propriétés du résultat fourni par la méthode. Un invariant définit des propriétés qui doivent vérifiées l'état de l'objet avant et après chaque appel de méthode.

Couplage : mesure la force de la relation entre deux modules, i.e. des classes dans le cas des modèles orientés objet.

Critère de test : critère qui doit être vérifié par des cas de test sur un programme sous test. Par exemple, couverture des chemins, couverture des prédicats, score de mutation…

Défaillance: événement survenant lorsque le service délivré dévie de l'accomplissement de la fonction du système.

Design Pattern : un design pattern représente une solution aux problèmes qui surgissent quand un logiciel est développé dans un contexte particulier. Les design patterns peuvent être considérés en tant que microarchitectures réutilisables qui contribuent à une architecture du système globale; ils saisissent les structures dynamiques et statiques et les collaborations entre les principaux participants du modèle de conception.

Délégation : mécanisme de programmation objet qui consiste, pour un objet, à déléguer tout ou partie du traitement d'une méthode à un autre objet.

Diagnosabilité : mesure de l'effort pour le diagnostic d'un programme.

Diagnostic : localisation et correction d'une faute lorsqu'un cas de test a échoué.

Donnée de test : donnée en entrée pour exécuter un programme sous test.

Double renvoi : double mécanisme de délégation.

Driver de test : programme qui exécute les cas de test sur le programme à tester, et qui sauvegarde les verdicts pour chaque cas de test.

Erreur: partie de l'état d'un système qui est susceptible d'entraîner une défaillance.

Faute: cause adjugée ou supposée d'une erreur.

Framework : Un framework est un ensemble de classes et de collaborations entre les instances de ces classes.

OCL : Object Constraint Language. Langage d'assertions pour l'expression de contraintes sur un modèle UML.

Oracle : une fonction d'oracle calcule la valeur attendue comme résultat d'un programme en fonction des données en entrée.

Orienté objets (conception) : la conception objet a pour but de préparer l'implantation d'un programme dans un langage orienté objet.

Polymorphisme : le fait qu'une référence puisse pointer sur des objets de types différents au cours de l'exécution.

Robustesse : La robustesse d'un composant est définie comme la capacité de ce composant à fonctionner même dans des conditions anormales. La robustesse peut aussi être vue comme la probabilité qu'une erreur soit détectée et traitée par un mécanisme d'exception sachant qu'une défaillance se produit certainement sinon.

Testabilité : La testabilité est un facteur de qualité, défini comme la facilité à tester un logiciel. La testabilité se dérive en trois attributs :

- le coût global de test : le coût global pour obtenir des cas de test vérifiant un critère de test donné
- la contrôlabilité : la facilité pour générer des données de test efficaces (propriété intrinsèque au logiciel sous test)
- l'observabilité : la facilité avec laquelle il est possible de vérifier la pertinence des résultats obtenus après l'exécution des cas de test

Test de logiciel : il existe deux grandes catégories de test logiciel :

le test dynamique qui consiste à exécuter des cas de test sur un programme et vérifier si la valeur obtenue est conforme à la valeur attendue.

le test statique qui observe les propriétés statiques d'un programme pour détecter des fautes.

UML : Unified Modelling Language. Langage graphique pour la conception de logiciels, en particulier des logiciels orientés objet.

Verdict de test : un cas de test *passe* si la valeur obtenue après exécution du cas de test est conforme à l'oracle, sinon il *échoue*.

Bibliographie

[Abdurazik"00] A. Abdurazik and A. J. Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation". In Proceedings of *UML'00*, York, UK. pp. 383 - 395, October 2000.

[AGEDIS"00] AGEDIS, "Automated Generation and Execution of Test Suites for DIstributed Component-based Software".

http://www.agedis.de/index.shtml.

[Agerbo"98] E. Agerbo and A. Cornils, "How to Preserve the Benefits of Design Patterns". In Proceedings of *OOPSLA'98*, Vancouver, BC, Canada. pp. 134 - 143, October 1998.

[Agrawal"95] H. Agrawal, J. Horgan, S. London and W. Wong, "Fault Localization using Execution Slices and Dataflow Tests". In Proceedings of *ISSRE'95 (Int. Symposium on Software Reliability Engineering)*, Toulouse, France. pp. 143 - 151, October 1995.

[Alexander"00] R. T. Alexander and J. Offutt, "Criteria for Testing Polymorphic Relationships". In Proceedings of *ISSRE'00 (Int. Symposium on Software Reliability Engineering)*, San Jose, CA, USA. pp. 15 - 23, October 2000.

[Alexander"02a] R. T. Alexander, A. J. Offutt and J. M. Bieman, "Fault Detection Capabilities of Coupling-Based OO Testing". In Proceedings of *ISSRE'02 (Int. Symposium on Software Reliability Engineering)*, Annapolis, MD, USA. pp. 207 - 218, November 2002.

[Alexander"02b] R. T. Alexander, J. M. Bieman, G. Sudipto and J. Bixia, "Mutation of Java Objects". In Proceedings of *ISSRE'02 (Int. Symposium on Software Reliability Engineering)*, Annapolis, MD, USA. pp. 341 - 351, November 2002.

[Antoniol"02] G. Antoniol, L. Briand, M. DiPenta and Y. Labiche, "A case Study Using The Round-Trip Strategy for State-Based Class Testing". In Proceedings of *ISSRE'02 (Int.*

Symposium on Software Reliability Engineering), Annapolis, MD, USA. pp. 269 - 279, November 2002.

[Baudry"00a] B. Baudry, Y. Le Traon and V. L. Hanh, "Testing-for-Trust: the Genetic Selection Model applied to Component Qualification". In Proceedings of *TOOLS Europe (Technology of object-oriented languages and systems)*, Mont St Michel, France. pp. 108 - 119, June 2000.

[Baudry"00b] B. Baudry, Y. Le Traon, V. L. Hanh and J.-M. Jézéquel, "Building Trust into OO Components using a Genetic Analogy". In Proceedings of *ISSRE'00 (Int. Symposium on Software Reliability Engineering)*, San Jose, CA, USA. pp. 4 - 14, October 2000.

[Baudry"00c] B. Baudry, Y. Le Traon, J.-M. Jézéquel and V. L. Hanh, "Trustable Components: Yet Another Mutation-Based Approach". In Proceedings of *1st Symposium on Mutation Testing*, San Jose, CA. pp. 69 - 76, October 2000.

[Baudry"01a] B. Baudry, Y. Le Traon and J.-M. Jézéquel, "Robustness and Diagnosability of Designed by Contracts OO Systems". In Proceedings of *Metrics'01 (Software Metrics Symposium)*, London, UK. pp. 272 - 283, April 2001.

[Baudry"01b] B. Baudry, Y. Le Traon, G. Sunyé and J.-M. Jézéquel, "Towards a Safe Use of Design Patterns for OO Software Testability". In Proceedings of *ISSRE'01 (Int. Symposium on Software Reliability Engineering)*, Hong-Kong, China. pp. 324 - 329, November 2001.

[Baudry"02a] B. Baudry, Y. Le Traon and G. Sunyé, "Testability Analysis of UML Class Diagram". In Proceedings of *Metrics'02 (Software Metrics Symposium)*, Ottawa, Canada. pp. 54 - 63, June 2002.

[Baudry"02b] B. Baudry, F. Fleurey, Y. Le Traon and J.-M. Jézéquel, "Automatic Test Cases Optimization using a Bacteriological Adaptation Model: Application to .NET Components". In Proceedings of *ASE'02 (Automated Software Engineering)*, Edimburgh, Scotland, UK. pp. 253 - 256, September 2002.

[Baudry"02c] B. Baudry, F. Fleurey, Y. Le Traon and J.-M. Jézéquel, "Computational Intelligence for Testing .NET Components". In Proceedings of *Microsoft Summer Research Workshop*, Cambrige, UK. pp. September 2002.

[Baudry"02d] B. Baudry, F. Fleurey, Y. Le Traon and J.-M. Jézéquel, "Genes and Bacteria for Automatic Test Cases Optimization in the .NET Environment". In Proceedings of *ISSRE'02 (Int. Symposium on Software Reliability Engineering)*, Annapolis, MD, USA. pp. 195 - 206, November 2002.

[Baudry"03] B. Baudry, Y. Le Traon, G. Sunyé and J.-M. Jézéquel, "Measuring and Improving Design Patterns Testability". In Proceedings of *Metrics'03 (Software Metrics Symposium)*, Sydney, Australia. pp. September 2003.

[Beck"99] K. Beck, "Extreme programming explained". Addison-Wesley 1999.

[Beck"01] K. Beck and E. Gamma, "JUnit".

http://www.junit.org/index.htm.

[Beizer"90] B. Beizer, "Software Testing Techniques". Van Norstrand Reinhold 1990.

[Bieman"89] J. M. Bieman and J. Schultz, "Estimating the Number of Test Cases Required to Satisfy the all-du-paths Testing Criterion". In Proceedings of *Software Testing Analysis and Verification Sympoium* pp. 179 - 186, December 1989.

[Binder"94] R. V. Binder, "Design for Testability in Object-Oriented systems". Communications of the ACM **37**(9): 87 - 101 September 1994.

[Binder"99] R. V. Binder, "Testing Object-Oriented Systems: Models, Patterns and Tools". Addison-Wesley 1999.

[Boden"96] E. B. Boden and G. F. Martino, "Testing software using order-based genetic algorithms". In Proceedings of *Genetic Programming 1996 Conference*, Stanford, CA, USA. pp. 461 - 466, July 1996.

[Bogdanov"01] K. Bogdanov and M. Holcombe, "Statechart Testing Method for Aircraft Control Systems". Software Testing, Verification and Reliability **11**(1): 39 - 54 March 2001.

[Booch"99] G. Booch, J. Rumbaugh and I. Jacobson, "The Unified Modeling Language User Guide". Addison-Wesley object technology series 1999.

[Briand"96] L. Briand, S. Morasca and V. S. Basili, "Property-based Software Engineering Measurement". IEEE Transactions on Software Enginnering **22**(1): 68 - 86 January 1996.

[Briand"99] L. Briand, J. W. Daly and J. K. Wüst, "A Unified Framework for Coupling Measurement in Object-Oriented Systems". IEEE Transactions on Software Enginnering **25**(1): 91 - 121 January/February 1999.

[Briand"01] L. Briand and Y. Labiche, "Revisiting Strategies for Ordering Class Integration Testing in the Presence of Dependency Cycles". In Proceedings of *ISSRE'01 (Int. Symposium on Software Reliability Engineering)*, Hong-Kong, China. pp. 287 - 296, December 2001.

[Briand"02a] L. Briand and Y. Labiche, "A UML-based approach to System Testing". Journal of Software and Systems Modeling **1**(1): 10 - 42 September 2002.

[Briand"02b] L. Briand, Y. Labiche and H. Sun, "Investigating the use of analysis contracts to support fault isolation in object oriented code". In Proceedings of *International Symposium on Software Testing and Analysis*, Roma, Italy. pp. 70 - 80, June 2002.

[Brown"98] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick, III and T. J. Mowbray, "AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis". John Wiley & Sons 1998.

[Buy"00] U. Buy, A. Orso and M. Pezzè, "Automated Testing of Classes". In Proceedings of *ISSTA'00*, Portland, OR, USA. pp. 39 - 48, August 2000.

[Carrillo-Castellon"96] M. Carrillo-Castellon, J. Garcia-Molina, E. Pimentel and I. Repiso, "Design by contract in Smalltalk". Journal of Object-Oriented Programming **8**(7): 23 - 38 November 1996.

[Chen"99] M.-H. Chen and H. M. Kao, "Testing object-oriented programs - an integrated approach". In Proceedings of *ISSRE'99 (Int. Symposium on Software Reliability Engineering)*, Boca Raton, FL, USA. pp. 73 - 82, November 1999.

[Cheon"02] Y. Cheon and G. T. Leavens, "A Simple and Practical Approach to Unit Testing: The JML and JUnit Way". In Proceedings of *ECOOP*, Malaga, Spain. pp. 231 - 255, June 2002.

[Chevalley"01a] P. Chevalley and P. Thévenod-Fosse, "Automated generation of statistical test cases from UML state diagrams". In Proceedings of *COMPSAC (25th Annual International Computer Software and Applications Conference)*, Chicago, IL, USA. pp. 205 - 214, October 2001.

[Chevalley"01b] P. Chevalley and P. Thevenod-Fosse, "An Empirical Evaluation of Statistical Testing Designed from UML State Diagrams: the Flight Guidance System Case Study". In Proceedings of *ISSRE'01 (Int. Symposium on Software Reliability Engineering)*, Hong Kong, China. pp. 27 - 30, November 2001.

[Chevalley"01c] P. Chevalley, "Applying Mutation Analysis for Object-Oriented Programs Using a Reflective Approach". In Proceedings of *Eighth Asia-Pacific Software Engineering Conference*, Macao, China. pp. 267 - 70, December 2001.

[Chidamber"94] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design". IEEE Transactions on Software Enginnering **20**(6): 476 - 493 June 1994.

[Clarke"76] L. A. Clarke, "A system to generate test data and symbolically execute programs". IEEE Transactions on Software Enginnering **2**(3): 215 - 22 September 1976.

[Collet"96] P. Collet and R. Rousseau, "Assertions are Objects too!" In Proceedings of *First Intern. Conf. on Object-Oriented Technology*, St Petersburgh, Russia. pp. June 1996.

[Correa"00] A. Correa, C. M. L. Werner and G. Zaverucha, "Object Oriented Design Expertise Reuse: An Approach Based on Heuristics, Design Patterns and Anti-patterns". In Proceedings of *International Conference on Software Reuse* pp. 336 - 352, June 2000.

[Craig"02] P. Craig, "NUnit".

http://nunit.sourceforge.net/.

[de Icaza"01] M. de Icaza, "mono : a C# compiler for Linux."

http://www.go-mono.com/index.html.

[DeMillo"78] R. DeMillo, R. Lipton and F. Sayward, "Hints on Test Data Selection : Help For The Practicing Programmer". IEEE Computer **11**(4): 34 - 41 April 1978.

[DeMillo"91] R. DeMillo and A. J. Offutt, "Constraint-Based Automatic Test Data Generation". IEEE Transactions on Software Enginnering **17**(9): 900 - 910 September 1991.

[DeMillo"93] R. DeMillo and A. J. Offutt, "Experimental results from an automatic test case generator". ACM Transactions on Software Engineering and Methodology **2**(2): 109 - 27 April 1993.

[Deveaux"99a] D. Deveaux, J.-M. Jézéquel and Y. Le Traon, "Self-testable Components: from Pragmatic Tests to a Design-for-Testability Methodology". In Proceedings of *TOOLS'99 (Technology of Object Oriented Languages and Systems)*, Nancy, France. pp. 96 - 107, June 1999.

[Deveaux"99b] D. Deveaux, J.-M. Jézéquel and Y. Le Traon, "Self-testable components: from pragmatic tests to design-for-testability methodology." In Proceedings of *Technology of Object Oriented Languages and Systems (TOOLS'99)*, Nancy, France. pp. 96-107, June 1999.

[Deveaux"01] D. Deveaux and Y. Le Traon, "XML to Manage Source Code Engineering in Object-Oriented Development: an Example." In Proceedings of *XSE01 workshop at ICSE'2001*, Toronto, Canada. pp. 28 - 31, May 2001.

[Dick"93] J. Dick and A. Faivre, "Automating the Generation and Sequencing of Test Cases from Model-based Specifications". In Proceedings of *FME'93* pp. 268 - 284, April 1993.

[D'Souza"98] D. F. D'Souza and A. C. Wills, "Object, Components and Frameworks with UML, The Catalysis Approach". Object Technology, Addison-Wesley 1998.

[Dustin"99] E. Dustin, J. Rashka and J. Paul, "Automated Software Testing". Addison-Wesley 1999.

[Eden"99] A. H. Eden "Precise Specification of Design Patterns and Tool Support in their Application". University of Tel Aviv, 1999.

[Edwards"01] S. H. Edwards, "A Framework for Practical, Automated Balck-Box Testing of Component-Based Software". Software Testing, Verification and Reliability **11**(2): 97 - 111 June 2001.

[Fenkam"02] P. Fenkam, H. Gall and M. Jazayeri, "Constructing Corba-Supported Oracles for Testing: A Case Study in Automated Software Testing". In Proceedings of *ASE'02 (Automated Software Engineering)*, Edimburgh, UK. pp. 129 - 138, September 2002.

[Fenton"86] N. E. Fenton and R. W. Whitty, "Axiomatic Approach to Software Metrication through Program Decomposition". Computer Journal **29**(4): 330 - 339 August 1986.

[Fewster"99] M. Fewster and D. Graham, "Software Test Automation". Addison-Wesley 1999.

[Findler"01] R. B. Findler and M. Felleisen, "Contract Soudness for Object-Oriented Languages". In Proceedings of *OOPSLA 2001*, Tampa Bay, FL, USA. pp. 1 - 15, October 2001.

[Fowler"97] M. Fowler, K. Scott and G. Booch, "UML distilled". Object Oriented series, Addison-Wesley 1997.

[Fowler"01] M. Fowler, "Reducing Coupling". IEEE Software **18**(4): 102 - 104 July-August 2001.

[Freedman"91] R. S. Freedman, "Testability of Software Components". IEEE Transactions on Software Enginnering **17**(6): 553 - 564 June 1991.

[Gamma"95] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software". Professional Computing, Addison-Wesley 1995.

[Ghosh"00] S. Ghosh and A. Mathur, "Interface Mutation to assess the adequacy of tests for components and systems". In Proceedings of *TOOLS*, Santa Barbara, CA, USA. pp. 37 - 46, August 2000.

[Goldberg"89] D. E. Goldberg, "Genetic Algorithms in Search, Optimization and Machine Learning". Addison-Wesley 1989.

[Grieskamp"02] W. Grieskamp, Y. Gurevich, W. Schulte and M. Veanes, "Generating Finite State Machines from Abstract State Machines". In Proceedings of *ISSTA'02 (International Symposium on Software Testing and Analysis)*, Rome, Italy. pp. 112-122, July 2002.

[Hanh"02] V. L. Hanh "Test et modèle UML : stratégie, plan et synthèse de test". Université de Rennes 1, 2002.

[Harrold"94] M. J. Harrold and G. Rothermel, "Performing Data Flow Testing on Classes". In Proceedings of *FSE (Foundation on Software Engineering)*, New Orleans, US. pp. 154 - 163, December 1994.

[Ho"99] W.-M. Ho, J.-M. Jézéquel, A. Le Guennec and F. Pennaneac'h, "UMLAUT: an Extendible UML Transformation Framework". In Proceedings of *ASE'99 (Automated Software Engineering)*, Cocoa Beach, Florida, USA. pp. 275 - 278, October 1999.

[Hoijin"98] Y. Hoijin, C. Byoungju and J. Jin-Ok, "Mutation-based inter-class testing". In Proceedings of *Asia Pacific Software Engineering Conference*, Taipei, Taiwan. pp. 174 - 181, December 1998.

[Holland"74] J. H. Holland, "Adaptation in Natural and Artificial Systems". University of Michigan Press 1974.

[Howden"70] W. E. Howden and Y. Huang, "Software Trustability". In Proceedings of *IEEE Symposium on Adaptive processes- Decision and Control* pp. 1970.

[Jéron"98] T. Jéron and P. Morel, "Test Generation Derived from Model-checking". In Proceedings of *CAV'99*, Kyoto, Japan. pp. 108 - 122, July 1998.

[Jézéquel"97] J.-M. Jézéquel and B. Meyer, "Design by Contract: The lessons of Ariane". Computer **30**(1): 129 - 130 January 1997.

[Jézéquel"99] J.-M. Jézéquel, M. Train and C. Mingins, "Design Patterns and Contracts". Addison-Wesley 1999.

[Jézéquel"01] J.-M. Jézéquel, D. Deveaux and Y. Le Traon, "Reliable Objects: a Lightweight Approach Applied to Java". IEEE Software **18**(4): 76 - 83 July/August 2001.

[Jones"96] B. F. Jones, H.-H. Sthamer and D. E. Eyres, "Automatic Structural Testing Using Genetic Algorithms". Software Engineering Journal **11**(5): 299 - 306 September 1996.

[Jones"98] B. F. Jones, D. E. Eyres and H.-H. Sthamer, "A Strategy for using Genetic Algorithms to Automate Branch and Fault-based Testing". The Computer Journal **41**(2): 98 - 107 1998.

[Jones"02] J. A. Jones, M. J. Harrold and J. Stasko, "Visualization of Test Information to Assist Fault Localization". In Proceedings of *ICSE'02*, Orlando, FL, USA. pp. 467 - 477, May 2002.

[Kamkar"95] M. Kamkar, "An Overview and Comparative Classification of Program Slicing Techniques". Journal of Systems and Software **31**(3): 197 - 214 December 1995.

[Kim"01] S.-W. Kim, J. A. Clark and J. A. McDermid, "Investigating the effectiveness of object-oriented testing strategies using the mutation method". Software Testing, Verification and Reliability **11**(4): 207 - 225 December 2001.

[Kim"99] Y. G. Kim, H. S. Hong, D. H. Bae and S. D. Cha, "Test cases generation from UML state diagrams". IEEE Proceedings-Software **146**(4): 187 - 192 August 1999.

[Kitchenham"95] B. Kitchenham, S. L. Pfleeger and N. Fenton, "Towards a framework for software measurement validation". IEEE Transactions on Software Enginnering **21**(12): 929 - 944 December 1995.

[Korel"90] B. Korel, "Automated Software Test Data Generation". IEEE Transactions on Software Enginnering **16**(8): 870 - 879 August 1990.

[Korel"92] B. Korel, "Dynamic method for software test data generation". Software Testing, Verification and Reliability **2**(4): 203 - 213 December 1992.

[Korel"96] B. Korel and A. M. Al-Yami, "Assertion-oriented automated test data generation". In Proceedings of *ICSE*, Berlin, Germany. pp. 71 - 80, March 1996.

[Korel"97] B. Korel, "Computation Of Dynamic Program Slices For Unstructured Programs". IEEE Transactions on Software Enginnering **23**(1): 17 - 34 January 1997.

[Kung"96] D. C. Kung, J. Gao, P. Hsia, Y. Toyashima and C. Chen, "On Regression Testing of Object-Oriented Programs". The Journal of Systems and Software **32**(1): 21 - 40 January 1996.

[Labiche"00] Y. Labiche, P. Thevenod-Fosse, H. Waeselynck and M.-H. Durand, "Testing levels for object-oriented software". In Proceedings of *ICSE*, Limerick, Ireland. pp. June 2000.

[Laprie"95] J.-C. Laprie, "Guide de la surete de fonctionnement". Cepadues 1995.

[Le Guennec"00] A. Le Guennec, G. Sunyé and J.-M. Jézéquel, "Precise Modeling of Design Patterns". In Proceedings of *UML'00* pp. 482 - 496, October 2000.

[Le Traon"97] Y. Le Traon and C. Robach, "Testability Measurements for Data Flow Designs". In Proceedings of *International Software Metrics Symposium (Metrics'97)*, Albuquerque, NM, USA. pp. 91 - 98, November 1997.

[Le Traon"98] Y. Le Traon, F. Ouabdessalam and C. Robach, "Software Diagnosability". In Proceedings of *ISSRE'98 (Int. Symposium on Software Reliability Engineering)*, Paderborn, Germany. pp. 257 - 266, November 1998.

[Le Traon"00a] Y. Le Traon, T. Jéron, J.-M. Jézéquel and P. Morel, "Efficient OO Integration and Regression Testing". IEEE Transactions on Reliability **49**(1): 12 - 25 March 2000.

[Le Traon"00b] Y. Le Traon, F. Ouabdessalam and C. Robach, "Analyzing Testability on Data Flow Designs". In Proceedings of *ISSRE'00 (Int. Symposium on Software Reliability Engineering)*, San Jose, CA, USA. pp. 162 - 173, October 2000.

[Le Traon"03] Y. Le Traon, F. Ouabdessalam, C. Robach and B. Baudry, "From Diagnosis to Diagnosability: Axiomatization, Measurement and Application". Journal of Systems and Software **65**(1): 31 - 50 January 2003.

[Legeard"02] B. Legeard, F. Peureux and M. Utting, "Automated Boundary Testing from Z and B". In Proceedings of *Formal Methods (FME'02)*, Copenhagen, Denmark. pp. 21 - 40, July 2002.

[Liskov"86] B. Liskov and J. Guttag, "Abstraction and Specification in Program Development". MIT Press/Mc Graw Hill 1986.

[Lugato"02] D. Lugato, C. Bigot and Y. Valot, "Validation and automaitc test generation on UML models: the AGATHA approach". Electronic Notes in Theoretical Computer Science **66**(2) December 2002.

[Ma"02] Y.-S. Ma, Y.-R. Kwon and A. J. Offutt, "Inter-Class Mutation Operators". In Proceedings of *ISSRE'02 (Int. Symposium on Software Reliability Engineering)*, Annapolis, MD, USA. pp. 352 - 363, November 2002.

[Marinov"01] D. Marinov and S. Khurshid, "TestEra: A Novel Framework for Automated Testing of Java Programs". In Proceedings of *ASE'01 (Automated Software Engineering)*, San Diego, CA, USA. pp. 22 - 31, November 2001.

[Martena"02] V. Martena, A. Orso and M. Pezzè, "Interclass Testing of Object Oriented Software". In Proceedings of *International Conference on Engineering of Complex Computer Systems* pp. 2002.

[McGregor"99] J. D. McGregor, "Test Patterns: Please Stand By". Journal of Object Oriented Programming **12**(3): 14 - 19 June 1999.

[McGregor"01] J. D. McGregor, "A Practical Guide To Testing Object Oriented Testing". Object Technology Series, Addison Wesley 2001.

[McKim"95] J. McKim, "Class Interface Design and Programming by Contract". In Proceedings of *TOOLS 17 (Technology of Object Oriented Languages and Systems)*, Englewood Cliffs. pp. 395 - 419, 1995.

[Meyer"92a] B. Meyer, "Object-oriented software construction". Prentice Hall 1992.

[Meyer"92b] B. Meyer, "Applying Design by Contract". IEEE Computer **25**(10): 40 - 51 October 1992.

[Michael"01] C. C. Michael, G. McGraw and M. A. Schatz, "Generating Software Test Data by Evolution". IEEE Transaction on Software Engineering **27**(12): 1085 - 1110 December 2001.

[Miller"76] W. Miller and D. L. Spooner, "Automatic Generation of Floating-Point Test Data". IEEE Transactions on Software Enginnering **2**(3): 223 - 226 September 1976.

[Mitchell"99] R. Mitchell and J. McKim, "Extending a method of devising software contracts". In Proceedings of *Tools'32* pp. 1999.

[MSDN"02a] MSDN, "C# Introduction and Overview".

http://msdn.microsoft.com/vstudio/techinfo/articles/upgrade/Csharpintro.asp.

[MSDN"02b] MSDN, ".NET homepage".

http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000519.

[Murray"98] L. Murray, D. Carrington, I. MacColl, J. McDonald and P. Strooper, "Formal Derivation of Finite State Machines for Class Testing". In Proceedings of *ZUM '98: The {Z} Formal Specification Notation*, Berlin, Germany. pp. 42-59, September 1998.

[Nebut"02] C. Nebut, S. Pickin, Y. Le Traon and J.-M. Jézéquel, "Reusable Test Requirements for UML-Modeled Product Lines". In Proceedings of *REPL*, Essen, Germany. pp. 51 - 56, September 2002.

[Nordby"02] J. E. Nordby, M. Blom and A. Brunstrom, "On the Relation between Design Contracts and Errors: a Software Development Strategy". In Proceedings of *ECBS*, Lund, Sweden. pp. April 2002.

[Offutt"92] A. J. Offutt, "Investigations of the software testing coupling effect". ACM Transactions on Software Engineering and Methodology **1**(1): 5 - 20 January 1992.

[Offutt"96a] A. J. Offutt, J. Pan, K. Tewary and T. Zhang, "An experimental evaluation of data flow and mutation testing". Software Practice and Experience **26**(2) February 1996.

[Offutt"96b] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch and C. Zapf, "An Experimental Determination of Sufficient Mutant Operators". ACM Transactions on Software Engineering and Methodology **5**(2): 99 - 118 April 1996.

[Offutt"97] A. J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths". The Journal of Software Testing, Verification and Reliability **7**(3): 165 - 192 September 1997.

[Offutt"99] A. J. Offutt and A. Abdurazik, "Generating Tests from UML Specifications". In Proceedings of *UML'99*, Fort Collins, CO, USA. pp. 416 - 429, October 1999.

[OMG"97] OMG, "Object Constraint Language Specification".

http://www.omg.org/docs/ad/97-08-08.pdf.

[Opdyke"92] W. F. Opdyke "Refactoring object-oriented frameworks". University of Illinois, 1992.

[Pargas"99] R. Pargas, M. J. Harrold and R. Peck, "Test-Data Generation Using Genetic Algorithms". Journal of Software Testing, Verifications, and Reliability **9**: 263 - 283 September 1999.

[Pickin"02] S. Pickin, C. Jard, Y. Le Traon, T. Jéron, J.-M. Jézéquel and A. Le Guennec, "System Test Synthesis from UML Models of Distributed Software". In Proceedings of *FORTE* pp. 2002.

[Rapps"85] S. Rapps and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information". IEEE Transactions on Software Enginnering **11**(4): 367 - 375 April 1985.

[Richner"99] T. Richner and S. Ducasse, "Recovering high-level views of object-oriented applications from static and dynamic information". In Proceedings of *ICSM'99 (International Conference on Software Maintenance)*, Oxford, UK. pp. 13 - 22, September 1999.

[Rosenblum"95] D. S. Rosenblum, "A Practical Approach to Programming With Assertions". IEEE Transactions on Software Enginnering **21**(1): 19 - 31 January 1995.

[Rosenzweig"95] M. L. Rosenzweig, "Species Diversity In Space and Time". Cambridge University Press 1995.

[Schultz"93] A. C. Schultz, J. J. Grefenstette and K. A. De Jong, "Test and Evaluation by Genetic Algorithms". IEEE Expert **8**(5): 9 - 14 October 1993.

[Shepperd"93] M. Shepperd and D. Ince, "Derivation and Validation of Software Metrics". Oxford University Press 1993.

[Shepperd"98] M. Shepperd, "Object-Oriented Metrics: an Annotated Bibliography".

http://dec.bournemouth.ac.uk/ESERG/bibliography.html.

[Shyam"94] S. R. Shyam and C. F. Kemerer, "A Metrics Suite for Object Oriented Design". IEEE Transactions on Software Enginnering **20**(6): 476 - 493 June 1994.

[Siegel"96] S. Siegel, "Object Oriented Software Testing". John Wiley & Sons 1996.

[Soundarajan"01] N. Soundarajan and B. Tyler, "Specification-based incremental testing of object oriented systems". In Proceedings of *TOOLS 39*, Santa Barbara, CA, USA. pp. 35 - 44, July 2001.

[SUN"00] SUN, "JavaCC".

http://www.webgain.com/products/java_cc/.

[SUN"02] SUN, "JavaTM Platform Debugger Architecture".

http://java.sun.com/products/jpda.

[Sunyé"00] G. Sunyé, A. Le Guennec and J.-M. Jézéquel, "Design Pattern Application in UML". In Proceedings of *ECOOP'00* pp. 44 - 62, June 2000.

[Szyperski"98] C. Szyperski, "Component Software: Beyond Object-Oriented Programming". ACM Press and Addison Wesley 1998.

[Tahat"01] L. H. Tahat, B. Vaysburg, B. Korel and A. J. Bader, "Requirement-based automated black-box test generation". In Proceedings of *COMPSAC*, Chicago, IL, USA. pp. 489 - 495, October 2001.

[Tai"99] K. C. Tai and F. J. Daniels, "Inter-Class Test Order for Object-Oriented Software". Journal of Object Oriented Programming **12**(4): 18 - 35 July-August 1999.

[Tracey"98] N. Tracey, J. Clark, K. Mander and J. A. McDermid, "An automated framework for structural test-data generation". In Proceedings of *ASE*, Honolulu, HI, USA. pp. 285 - 288, October 1998.

[Tracey"00] N. Tracey, J. Clark, K. Mander and J. McDermid, "Automated test-data generation for exception conditions". Software Practice and Experience **30**(1): 61 - 79 January 2000.

[Tsai"01] W.-T. Tsai, X. Bai, R. Paul and L. Yu, "Scenario-based functional regression testing". In Proceedings of *COMPSAC 2001*, Chicago, IL, USA. pp. 496 - 501, October 2001.

[Voas"92a] J. M. Voas and K. Miller, "The Revealing Power of a Test Case". Software Testing, Verification and Reliability **2**(1): 25 - 42 May 1992.

[Voas"92b] J. M. Voas, "PIE : A Dynamic Failure-Based Technique". IEEE Transactions on Software Enginnering **18**(8): 717 - 727 August 1992.

[Voas"93] J. M. Voas and K. Miller, "Semantic Metrics for Software Testability". Journal of Systems and Software **20**(3): 207 - 216 March 1993.

[Voas"95] J. M. Voas and K. Miller, "Software Testability: The New Verification". IEEE Software **12**(3): 17 - 28 May 1995.

[Voas"99] J. M. Voas and L. Kassab, "Using Assertions to Make Untestable Software More Testable". Software Quality Professional **1**(4) September 1999.

[Wegener"01] J. Wegener, A. Baresel and H. Stahmer, "Evolutionary Test Environment for Automatic Structural Testing". Information and Software Technology **43**(14): 841 - 854 December 2001.

[Weide"96] B. W. Weide, S. H. Edwards, W. D. Heym, T. J. Long and W. F. Ogden, "Characterizing Observability and Controllability of Software Components". In Proceedings of *4th International Conference on Software Reuse*, Orlando, USA. pp. 62 - 71, April 1996.

[Weiser"82] M. Weiser, "Programmers Use Slices When Debugging". Communication of ACM **25**(7): 446 - 452 July 1982.

[Weiser"84] M. Weiser, "Program Slicing". IEEE Transactions on Software Enginnering **10**(4): 352 - 357 July 1984.

[Wu"03] Y. Wu, M.-H. Chen and A. J. Offutt, "UML-based Integration testing for Component-based Software". In Proceedings of *International Conference on COTS-based Software Systems*, Ottawa, Canada. pp. February 2003.

[Xanthakis"92] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas and K. Karapoulios, "Genetic Algorithms Applications to Software Testing". In Proceedings of *Fifth International Conference. Software Engineering and Its Applications*, Toulouse, France. pp. 625 - 636, December 1992.

Résumé : Le test de logiciel apparaît aujourd'hui comme le moyen principal pour la validation du fonctionnement d'un programme. Les travaux présentés au cours de cette thèse s'articulent autour de trois contributions qui se concentrent sur certaines spécificités de la programmation et de la conception orientée objet pour l'étude de solutions particulières pour le test d'un composant. La qualité des cas de test étant un facteur important pour une validation efficace, le premier point concerne l'étude d'algorithmes évolutionnistes pour la génération automatique de cas de test pour un composant. Nous nous intéressons ensuite à l'impact de la conception par contrat sur deux aspects du test d'un assemblage de composants : la détection et la localisation d'erreur. Enfin, certaines configurations sont identifiées sur un diagramme de classes, comme difficiles à tester, et des solutions sont proposées pour résoudre ces problèmes dés la conception et éviter ainsi d'implanter un programme peu testable.

Mots clés : test de logiciel, UML, conception par contrat, génération automatique de test, algorithmes évolutionnistes, testabilité, analyse de mutation, mesure du logiciel, design patterns, conception et analyse orientée objet.

Abstract: Software testing is a very important technique to validate a program. In this thesis, we deal with three different aspects of the validation of software components through the analysis of specific object-oriented features and design methods.

The first point we are interested in is the automatic improvement and generation of test cases. We study evolutionist algorithms, and propose an original algorithm called bacteriologic algorithm, efficient and adaptable for test generation. Secondly, we study the impact of design by contract for improving two testing activities: fault detection (robustness) and fault localization (diagnosability). The last point we investigate concerns patterns in a UML class diagram which can lead to hard-to-test programs. We propose a test criterion to cover these patterns and several possibilities to clarify the class diagram, and avoid testing problems when implementing it. We also study the specific impact of design patterns applications on testing.

Keywords: software testing, UML, design by contract, automated test generation, evolutionist algorithms, testability, mutation analysis, software measurement, design patterns, object-oriented design and analysis.