Chapitre 3

Modèles de qualité et conception des programmes

3.1. Introduction

Les programmes par objets sont présents aujourd'hui partout dans notre société et sont de plus en plus complexes. La tendance courante d'externaliser le développement et la maintenance des programmes nécessite de mesurer leur qualité avec beaucoup de détails et de précision. Par exemple, un de nos partenaires industriels, qui travaille dans le domaine ferroviaire et utilise de nombreux programmes par objets aussi bien en interne qu'à destination de ses clients, recherchait des modèles pour mesurer la qualité du code source livré par ses sous-traitants comme moyen de contrôler le coût de ces programmes et de contractualiser une qualité minimale.

De nombreux modèles de qualité existent dans la littérature. Cependant, comme l'avait remarqué Briand et Wust [BRI 02b] dans leur vue d'ensemble sur les modèles de qualité en 2002 et confirmé par l'état de l'art présenté dans la section 3.2, aucun modèle existant ne considère l'organisation des classes et n'étudie les avantages ou inconvénients d'utiliser cette organisation, c'est-à-dire la conception des programmes, dans sa mesure de la qualité. La *conception* d'un programme par objets correspond à l'organisation locale et implicite de ses classes, en opposition à son *architecture* qui est son organisation globale et qui peut être implicite ou explicite [EDE 03].

Chapitre rédigé par Foutse KHOMH, Yann-Gaël GUÉHÉNEUC, Giuliano ANTONIOL et Massimiliano DI PENTA.

La conception des programmes est l'un des premiers aspects des programmes que les développeurs doivent maîtriser avant de réaliser toute activité de développement ou de maintenance, y compris les activités liées à la programmation. Une conception est le résultat, intentionnel ou non, des besoins implicites et locaux des développeurs [EDE 03], traduits en groupes de classes avec des organisations spécifiques. Des organisations spécifiques reconnues sont décrites par les anti-patrons et patrons de conception. Ces organisations sont au cœur de notre méthode DEQUALITE et de son instantiation, PQMOOD, présentées dans ce chapitre.

Les patrons de conception [GAM 94] décrivent de « bonnes » solutions à des problèmes récurrents de conception par objets. Ils sont utilisés par les développeurs pour concevoir [BEC 94] ou améliorer par superposition la conception des programmes [HAN 02]. Les avantages des patrons de conceptions incluent une meilleure réutilisation, compréhension et maintenabilité. L'utilisation de patrons de conception rend aussi les programmes plus robustes aux changements, comme le remarquent Gamma *et al.* : « Each design pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change »[1] [GAM 94]. En pratique, les patrons de conception offrent des motifs de conception [GUÉ 08] qui sont des solutions idéales décrivant les rôles et l'organisation des classes qui implantent ces motifs. Des occurrences de motifs de conception sont présentes dans tout programme raisonnablement bien conçu [GAM 94].

Les anti-patrons affectent négativement la qualité et l'évolution des programmes [BRO 98]. Ce sont de « mauvaises » solutions à des problèmes récurrents de conception. Par exemple, les quarante anti-patrons proposés par Brown décrivent des problèmes reconnus dans le développement de programmes par objets. La présence d'occurrences d'anti-patrons est généralement due au manque de connaissances des développeurs ou à leur manque d'experience dans la résolution de problèmes de conception et dans leur « mauvaise » utilisation de patrons de conception, comme l'indique Coplien : « something that looks like a good idea, but which backfires badly when applied »[2] [COP 05]. En pratique, les anti-patrons sont liés et se manifestent dans le code source sous la formes de « mauvaises odeurs », symptômes de problèmes d'implantation et de conception [FOW 99].

Les patrons et anti-patrons de conception concernent l'organisation de plusieurs classes. Dans nos travaux précédents [KHO 09a, KHO 09b, JEA 09, ABB 11], nous montrons que les patrons et anti-patrons affectent significativement les propensions aux changements et aux fautes des classes des programmes par objets. Partant de ces résultats, ce chapitre surmonte trois défis :

---

1. Chaque patron de conception permet à un aspect de la structure du système de varier, rendant ainsi le système plus résistant à un type de changement particulier.

2. Quelque chose qui semble être une bonne idée mais qui tourne vraiment mal quand appliqué.

– le choix des caractéristiques de qualité à évaluer ;

– la création et la paramétrisation du modèle ;

– la validation du modèle obtenu.

Nous répondons à ces défis par un état de l'art sur la qualité et les anti-patrons et patrons de conception ; par une méthode de construction de modèles de qualité ; et, par une validation pour répondre aux questions de recherche :

– QR1 : avec quelle précision et quel rappel notre modèle peut-il évaluer la propension aux changements et aux fautes des classes d'un programme ?

– QR2 : est-ce que les résultats de notre modèle de qualité incluant la conception sont meilleurs que les résultats sans prendre en compte la conception des classes ?

Ainsi, ce chapitre présente un état de l'art sur les modèles de qualité. Il introduit ensuite notre méthode DEQUALITE et son instantiation, PQMOOD, pour la mesure de la propension aux changements et aux fautes des classes. Identifier les classes les plus susceptibles de changer et d'avoir des fautes que les autres est intéressant pour focaliser les efforts des développeurs lors des revues de code, de la vérification et de la validation des programmes et autres activités de maintenance. De nombreuses approches de mesure des classes plus sujettes aux changements et aux fautes existent dans la littérature, par example [BRI 02a, KOR 07] ; la spécificité de notre méthode et du modèle résultant est la prise en compte de la conception des programmes. Nous montrons au travers de PQMOOD et de sa validation qu'en plus des attributs internes des classes, leurs organisations aussi sont importantes pour évaluer leur qualité. Ainsi, nous concluons que lier les propensions des classes aux changements et aux fautes avec les anti-patrons et patrons de conception plutôt qu'avec seulement des valeurs de métriques de classes donne au modèle PQMOOD de meilleurs précisions et rappels que les modèles précédents.

Ce chapitre est organisé comme suit. La section 3.2 présente l'état de l'art. La section 3.3.1 décrit notre méthode DEQUALITE, la Section 3.3.2 présente son instantiation, PQMOOD et la section 3.3.3 rapporte les résultats de PQMOOD sur trois programmes. La section 3.3.4 discute les résultats et compare PQMOOD avec deux modèles précédents. La section 3.4 conclut et propose des travaux futurs.

3.2. État de l'art

Cette section présente les travaux de la littérature qui ont proposé des méthodes ou des modèles pour mesurer des caractéristiques de qualité des programmes. Dans ces travaux, la qualité est définie par un ensemble de caractéristiques de qualité, par exemple la flexibilité ou la propension aux changements. Ces caractéristiques sont décrites en termes d'attributs internes des classes et–ou des programmes ; eux-mêmes

mesurés à l'aide d'un ensemble de métriques et d'un ensemble de relations liant les valeurs de ces métriques et les caractéristiques de qualité. La plupart des modèles de qualité sont des modèles hiérarchiques pour éviter tout chevauchement entre attributs internes et caractéristiques de qualité.

3.2.1. *Méthode de construction de modèles de qualité*

Avec l'objectif de construire des modèles de qualité adaptés à différents types de programmes, Dromey proposa une méthode de construction de modèles de qualité qui reconnaît que « a more dynamic idea for modelling the process [of quality modelling] is needed to be wide enough to apply for different systems »[3] [DRO 95, DRO 96]. Bien qu'inspiré par le modèle de qualité de McCall *et al.* [MCC 77], Dromey argumente que la mesure de la qualité peut être différente pour différents programmes.

Avec sa méthode, Dromey se concentre sur les relations entre caractéristiques de qualité et essaie de les lier systématiquement à des attributs internes mesurables des programmes. Les principaux éléments de la méthode de Dromey sont :

– les caractéristiques de qualité considérées ;

– les attributs internes des programmes ;

– les relations entre attributs et caractéristiques de qualité.

La méthode de Dromey inclut un processus de construction de modèles de qualité en cinq étapes :

– choisir un ensemble de caractéristiques de qualité ;

– énumérer les composants/modules du programme à évaluer ;

– identifier les attributs internes des composants/modules qui affectent le plus les caractéristiques choisies ;

– déterminer comment chaque attribut affecte les caractéristiques ;

– évaluer le modèle ainsi créé et identifier ses faiblesses.

La méthode de Dromey préconise huit caractéristiques de qualité : les six du standard ISO/IEC 9126-1 ainsi que la réutilisabilité et la maturité du processus de développement du programme mesuré [VIN 07].

---

3. Une idée plus dynamique pour décrire le processus de modélisation de la qualité est nécéssaire pour s'appliquer à différents systèmes.

3.2.2. *Modèles de qualité*

Il existe de nombreux modèles de qualité, nous rapportons ici ceux qui sont les plus cités dans la littérature dont nous avons connaissance. D'autres modèles existent, tels que ceux de Evans et Marciniak [EVA 87], Deutsch et Willis [DEU 88] ou encore celui du standard IEEE 1061-1998 [IEE 98]. En général, il existe trois catégories de modèles de qualité suivant que les modèles permettent de définir, d'évaluer et–ou de prédire des caractéristiques de la qualité logicielle [DEI 09].

3.2.2.1. *McCall et al. (définition)*

Le premier modèle de qualité a été introduit par McCall *et al.* [MCC 77], qui travaillait alors au *US Air Force Electronic System Decision Department*. L'objectif de ce modèle est de définir les relations entre les attributs internes des programmes et leurs caractéristiques de qualité externes. McCall *et al.* ont sélectionné les attributs et les caractéristiques de qualité qui reflètent les points de vue d'usagers et de développeurs.

Ainsi, trois caractéristiques principales sont considérées dans la définition et la sélection des attributs internes et de leurs relations : révisions, transitions et opérations. Les révisions incluent comme caractéristiques la capacité des programmes à changer, leur maintenabilité, flexibilité et testabilité. Les transitions portent sur la portabilité, la réutilisabilité et l'interopérabilité des programmes. Les opérations portent sur la correction, la fiabilité, l'efficacité, l'intégrité et l'utilisabilité des programmes.

Le modèle de McCall organise chacune de ces trois caractéristiques principales en une hiérarchie de facteurs, critères et, finalement, métriques. McCall *et al.* proposent aussi un ensemble de métriques, des échelles de mesure et une méthode pour mesurer la qualité des programmes. Certaines métriques sont évaluées à l'aide de questionnaires administrés aux développeurs ou usagers des programmes.

La contribution principale de ce modèle est d'avoir été le premier modèle mettant en relation caractéristiques de qualité et métriques. Cependant, à cause de ses mesures subjectives de la qualité auprès des développeurs et des usagers, le modèle de McCall a été critiqué et peu utilisé en pratique.

3.2.2.2. *Boehm (définition)*

Se basant sur le modèle de McCall, Boehm [BOE 76, BOE 78] a proposé un modèle de qualité similaire à celui de McCall *et al.* en ce qu'il organise aussi les attributs internes et les caractéristiques de qualité en une hiérarchie ; avec des caractéristiques, des attributs internes et des métriques. Ce modèle cible différents types d'usagers qui travaillent avec le programme [VIN 07].

Le modèle de Boehm ajoute des caractéristiques supplémentaires au modèle de McCall *et al.*, mettant l'emphase sur la maintenabilité et sur les performances du matériel sur lequel le programme est exécuté. La caractéristique principale d'utilité générale est décomposée en portabilité, utilité et maintenabilité ; caractéristiques qui correspondent aux requis de haut niveau du programme qui est mesuré. La caractéristique d'utilité est de plus raffinée en fiabilité, efficacité et utilisabilité. La maintenabilité est raffinée en testabilité, compréhensibilité et changeabilité.

Alors que le modèle de McCall *et al.* se focalise principalement sur la mesure des attributs internes de qualité, le modèle de Boehm explore un plus large spectre de caractéristiques, mettant l'emphase sur la maintenabilité.

3.2.2.3. *ISO/IEC 9126-1 (définition)*

En 1991, dans un effort pour standardiser l'évaluation de la qualité des programmes, l'Organisation internationale de standardisation (ISO) proposa le standard ISO 9126, qui se divise en quatre parties : un modèle de qualité ; des caractéristiques de qualité, des attributs internes et des attributs internes en utilisation [ISO91].

ISO 9126 partie I, souvent référencée comme ISO/IEC 9126-1, spécifie six caractéristiques pour mesurer la qualité des programmes : fonctionnalité, fiabilité, utilisabilité, efficacité, maintenabilité et portabilité. Il fournit, pour chacune de ces caractéristiques, des relations et attributs internes pour les rendre mesurables.

La principale limitation de ce modèle est qu'il ne décrit pas explicitement comment les attributs internes de qualité doivent être mesurés. Cependant, il est souvent cité et de nombreuses entreprises l'utilisent comme base à leurs mesures de la qualité des programmes car il leur laisse toute latitude quant aux choix des métriques et des relations précises entre métriques, attributs internes et caractéristiques de qualité.

Ce standard est désormais en voie d'être remplacé par le standard ISO 25000 et ses implantations, qui tentent d'apporter plus de détails dans la modélisation et l'évaluation de la qualité [ISO 05].

3.2.2.4. *QMOOD de Bansiya et Davis (évaluation)*

En utilisant la méthode de construction de Dromey [DRO 95, DRO 96], Bansiya et Davis [BAN 02] proposèrent QMOOD, un modèle de qualité pour mesurer la qualité des programmes par objets.

QMOOD consiste en six équations qui établissent les relations entre six caractéristiques de qualité (réutilisabilité, flexibilité, compréhension, fonctionnalité, extensibilité et efficacité) et onze attributs internes des classes, parmi lesquelles : encapsulation, couplage, polymorphisme, abstraction des données et héritage. Les auteurs introduisent aussi un ensemble de métriques orientées objets pour mesurer ces attributs internes de conception.

Bansiya et Davis ont validé QMOOD sur trois programmes industriels en faisant l'hypothèse que leurs caractéristiques de qualité se sont améliorées au cours du temps : *Microsoft Foundation Classes* (cinq versions), *Borland Object Windows Library* (quatre versions) et quatorze versions d'un programme de taille moyenne écrit en C++ et implantant un interpréteur pour un langage nommé COOL.

Un des principaux avantages du modèle QMOOD est qu'il est facile à modifier pour inclure différentes relations et poids[4]. Les auteurs pensent ainsi fournir un outil pratique et concrete d'évaluation de la qualité adaptable à une grande variété d'usages[5]. QMOOD est le modèle de qualité le plus référencé dans la littérature.

3.2.2.5. *Zimmermann et al. (prédiction)*

Zimmermann *et al.* [ZIM 07] ont proposé un modèle pour prédire des aspects de la qualité de programmes. Ce modèle n'est pas un modèle de qualité comme les précédents en cela qu'il n'essaie pas d'évaluer une caractéristique de qualité définie, par exemple par ISO 9126 [ISO91], mais qu'il prédit les nombres de fautes pré- et post-distributions des classes d'un programme par objets à partir des valeurs de mesures de la complexité et de la taille des classes.

Nous citons ce travail car il s'agit d'un des premiers modèles de qualité à traiter de la propension aux fautes des classes de programmes par objets. Ce modèle a inspiré de nombreux travaux sur la propension aux fautes, en particulier dans une série d'ateliers de grande qualité, PROMISE. Nous utilisons aussi la propension aux fautes dans notre modèle de qualité présenté dans la suite de ce chapitre.

Le modèle de Zimmermann *et al.* est aussi un des premiers modèles construits à partir d'un programme de taille industriel, en l'occurrence Eclipse, pour non seulement définir le modèle mathématique sous-jacent mais aussi pour identifier les valeurs des constantes du modèle. La taille du programme Eclipse et l'effort d'analyse effectués par les auteurs ont inspiré la communauté à suivre leur exemple.

3.2.3. *Patrons de conception et qualité*

Depuis leur introduction par Gamma *et al.* [GAM 94], les patrons de conception ont reçu un intérêt croissant de la part de la communauté. De nombreux travaux portent sur ces patrons, depuis leur définition [KAM 07] jusqu'à leur identification [GUÉ 08]. Nous présentons ici chronologiquement des travaux sur l'impact des motifs de conception (c'est-à-dire les solutions concrètes proposées par les patrons de conception) sur la qualité des programmes par objets.

---

4. « [QMOOD can] be easily modified to include different relationships and weights ».

5. « [QMOOD is a] practical quality assessment tool adaptable to a variety of demands ».

Lange et Nakamura [LAN 95] ont montré que les patrons de conception peuvent servir de guide dans l'exploration de programmes et, ainsi, rendre le processus de compréhension plus efficace. Avec l'exploration dirigée par les patrons, ils ont montré que les motifs reconnus à un moment du processus de compréhension aident les développeurs à remplir « des trous » dans leurs modèles mentaux et à explorer plus avant le programme, facilitant ainsi leur compréhension. Cette étude est limitée aux motifs de conception composite, décorateur et observateur.

Wydaeghe *et al.* [WYD 98] ont présenté une étude sur l'utilisation concrète de six motifs de conception dans l'implantation d'un éditeur OMT. Ils ont discuté l'impact de ces motifs sur la réutilisabilité, la modularité, la flexibilité et la compréhension de l'éditeur. Ils ont aussi discuté la difficulté d'implanter concrètement les motifs. Ils ont conclu que tous les motifs de conception n'ont pas un impact positif sur la qualité interne des programmes, même si les patrons de conception offrent de nombreux avantages. Cette étude est limitée à l'expérience et l'évaluation subjective des auteurs et ses résultats ne sont donc pas généralisables.

Masuda *et al.* [MAS 99] ont aussi étudié l'impact de l'utilisation de motifs de conception dans des programmes sur leurs qualités internes. Ils ont implanté un ensemble de programmes avec deux versions pour chaque programme : une avec des motifs de conception ; une sans. Ils ont montré qu'il n'y a pas de différences statistiquement significatives entre les valeurs des métriques objets de Chidamber et Kemerer calculées sur les programmes avec ou sans motifs. Ils ont suggéré que de nouvelles métriques sont nécessaires pour mesurer les programmes avec des motifs.

Wendorff [WEN 01] a évalué l'utilisation de motifs de conception dans de grands programmes commerciaux. Il a introduit deux catégories de « mauvaise » implantation de motifs de conception. Dans la première catégorie, les motifs étaient utilisés à mauvais escient par les développeurs qui n'avaient pas compris leurs objectifs. Dans la seconde catégorie, les motifs étaient utilisés correctement par les développeurs mais sans réel besoin par rapport aux fonctionnalités des programmes. Ils ont analysé cette seconde catégorie et ont trouvé que ces utilisations étaient dues :

– aux développeurs surestimant les futurs besoins en changement et introduisant des motifs pour obtenir une plus grande flexibilité ;

– à des changements dans les besoins qui rendaient certains motifs obsolètes ;

– à l'utilisation de motifs sans égards pour la qualité des programmes (par exemple pour apprendre un patron) ;

– à des changement dans l'implantation des motifs pour les rendre plus proches des motifs suggérés dans le livre de Gamma *et al.*, à nouveau sans considération pour la qualité des programmes.

Ils ont conclu que les motifs de conception n'améliorent pas nécessairement la conception et l'implantation d'un programme et qu'un programme peut être *over-engineered*[6] [KER 04] et que le coût de supprimer un motif de conception est élevé.

Bieman *et al.* [BIE 01, BIE 03] ont étudié l'utilisation de styles recommandés de programmation, incluant des patrons de conception, dans différents programmes et ont conclu que, contrairement à ce qui était généralement admis, l'utilisation de patrons peut amener les classes à changer plus lors de l'évolution des programmes. Avec McNatt, Bieman a aussi réalisé une étude qualitative sur le couplage entre motifs [MCN 01]. Ils ont conclu que lorsque les classes jouant des rôles dans les motifs sont faiblement couplées et abstraites alors elles ont une bonne maintenabilité, modularité et réutilisabilité. Ils ont néanmoins identifié le besoin de plus d'études pour examiner la composition de différents motifs et leurs impacts sur la qualité.

Hannemann et Kiczales [HAN 02] ont étudié les motifs de conception et leur implantation à l'aide d'aspects. Ils ont montré que 17 des 23 motifs de conception bénéficient de leurs « aspectisation », qui permet de diminuer :

– l'impact des motifs sur les programmes ;

– l'impact des autres classes sur les classes jouant des rôles dans les motifs ;

– la perte de modularité et de traçabilité des motifs ;

– le côté invasif des motifs ;

– la difficulté de raisonner sur les classes impliquées dans plusieurs motifs.

Ils ont proposé des implantations AspectJ des motifs qui mettent en correspondance les dépendances dans le code avec celles dans la structure de la solution[7].

Vokac [VOK 04] a analysé la maintenance corrective d'un grand programme commercial. Il a identifié automatiquement les occurrences de motifs de conception contenues dans les versions hebdomadaires du programme sur une période de trois ans et a comparé les taux de fautes des classes jouant des rôles dans ces occurrences avec ceux des autres classes. En utilisant la régression logistique, il a rapporté que les classes jouant des rôles dans des motifs étaient moins sujettes à des fautes avec des différences de 63 % à 154 %, en moyenne. Il a aussi noté que :

– les motifs observateur et singleton sont corrélés à de grandes classes ;

– les classes jouant des rôles dans le motif méthode usine étaient moins cohésives, moins couplées et moins sujettes à des fautes que les autres classes ;

– aucune tendance claire n'était visible pour le motif méthode gabarit.

---

6. Sur-conçu.

7. « [Aspect-oriented solutions that] better align dependencies in the code with dependencies in the solution structure ».

Ces résultats fournissent des données quantitatives sur la relation entre motifs de conception et la propension aux fautes des classes jouant des rôles dans ces motifs.

Ng *et al.* [NG 07] ont étudié l'utilisation des motifs de conception par les développeurs pour les tâches suivantes :

– ajouter une nouvelle classe jouant un rôle dans un motif de conception (T1) ;

– modifier l'interface publique d'une classe jouant un rôle dans un motif (T2) ;

– introduire une nouvelle classe cliente (T3).

Leur étude s'est déroulée dans le contexte de la maintenance perfective car les auteurs pensaient qu'il s'agissait de l'activité de maintenance la plus commune et car selon eux la réalisation d'un changement anticipé implique une ou plusieurs des trois tâches T1–3. Leur expérimentation incluait 215 sujets qui devaient réaliser six changements anticipés dans trois programmes, avec un total de 17,8 KLOC et plus de 230 classes dans douze paquetages. Elle impliquait six motifs de conception qui couvraient les catégories créationelles, structurelles et comportementales. Les résultats de l'expérimentation montrent que tous les sujets ont réalisé les tâches T1, une majorité d'entre eux les tâches T3 mais, qu'en moyenne, moins de la moitié les tâches T2. Ils montrent aussi que le code implanté par les sujets qui ont utilisé les motifs de conception présents dans les programmes pour réaliser leurs tâches contenait significativement moins de fautes que le code des sujets qui n'avaient pas utilisé les motifs présents. Ils suggèrent que les motifs de conception réduisent la propension aux fautes.

Di Penta *et al.* [Di 08] ont étudié la propension aux changements de classes jouant différents rôles dans des motifs de conception et les types de changements affectant ces classes. Leurs résultats confirment l'impact attendu théorique des motifs de conception. Par exemple, ils ont trouvé que, dans le motif usine abstraite, les classes jouant des rôles concrets changent plus souvent que les classes jouant des rôles abstraits. Ils ont aussi identifié des différences avec l'impact attendu théorique, par exemple dans le cas du motif composite, dans lequel les classes jouant le rôle de composite peuvent être complexes et subir de nombreux changements. Dans un autre travail [AVE 07], ils ont étudié la résilience aux changements des classes jouant des rôles dans des motifs et ont conclu que ces classes changent fréquemment et que la quantité de changements ne dépend pas du motif mais du rôle joué par la classe pour soutenir les fonctionnalités du programme. Ils ont aussi observé que l'implantation des classes jouant des rôles dans les motifs changent plus souvent que leurs interfaces et que des sous-classes sont souvent ajoutées à ces classes.

3.2.4. *Anti-patrons et qualité*

Brown [BRO 98] a décrit les anti-patrons comme le résultat du manque de connaissance ou d'expérience des développeurs dans la résolution d'un problème particulier

de conception ou comme le résultat de la mauvaise application d'un patron de conception. Il a suggéré que les anti-patrons rendent la maintenance plus difficile et diminue la qualité des programmes. Il n'a cependant fourni aucune donnée quantitative soutenant ces affirmations.

Les premiers travaux qui ont étudié quantitativement la relation entre anti-patrons et qualité sont ceux de Deligiannis *et al.* [IGN 03, IGN 04]. Dans ces travaux, les auteurs ont réalisé des expériences avec vingt sujets sur deux programmes pour comprendre l'impact des *Blobs* sur la compréhension et la maintenance des programmes. Les résultats de ces travaux suggèrent que le *Blob* affecte négativement l'évolution de la conception des programmes et très fortement l'utilisation de l'héritage par les sujets. Du Bois *et al.* [BOI 06] ont décrit ensuite que la décomposition de *Blobs* en classes collaborantes, par des *refactorings* bien connus, améliorait la compréhension de ces classes par les développeurs.

Moha *et al.* [MOH 06, MOH 08, MOH 10] ont fourni trois contributions à la détection de « mauvaises odeurs » et d'anti-patrons dans le code de programmes par objets :

– la méthode DECOR qui décrit et définit les étapes nécessaires à la spécification et à la détection des mauvaises odeurs et des anti-patrons ;

– la technique de détection DEX issue de la méthode ;

– une validation empirique de DEX en termes de précision et de rappel.

L'originalité de DEX réside dans la possibilité qu'ont les développeurs de spécifier les mauvaises odeurs et les anti-patrons à un haut niveau d'abstraction avec un vocabulaire systématique et un langage spécifique et de générer ensuite automatiquement les algorithmes de détection. Avec DEX, les auteurs ont spécifié quatre anti-patrons : *Blob*, décomposition fonctionnelle, *Spaghetti Code* et couteau suisse et leur quinze mauvaises odeurs. Ils ont automatiquement généré les algorithmes de détection correspondants et les ont validés sur Xerces v2.7.0, rapportant un rappel de 100 % et une précision moyenne de 33 %.

Li et Shatnawi [WEI 07] ont étudié la propension aux fautes de classes jouant ou non des rôles dans des anti-patrons en utilisant trois versions d'Eclipse. Ils ont ainsi montré que les classes participant dans des *Blobs*, ayant subit une chirurgie au fusil de chasse (*Shotgun Surgery*) ou ayant des méthodes longues avaient une plus grande probabilité d'être sujettes à des fautes que les autres. Ils ont conclu sur le besoin d'études plus étendues et exhaustives pour confirmer leurs résultats.

Olbrich *et al.* [OLB 09] ont analysé les données historiques de Lucene et Xerces sur plusieurs années et ont conclu que les classes participant dans des *Blobs* et chirurgies au fusil de chasse (*Shotgun Surgery*) ont des fréquences de changements plus importantes que les autres ; les *Blobs* changeant le plus.

Khomh *et al.* [KHO 11] ont proposé l'utilisation des réseaux bayésiens pour la détection d'anti-patrons. Contrairement aux méthodes à base de règles, leur approche permet de calculer pour chaque classe d'un programme, et pour chaque anti-patron, la probabilité que la classe participe dans cet anti-patron, permettant ainsi de prendre en compte l'incertitude liée à la detection.

Vaucher *et al.* [VAU 09] ont étudié les causes de l'introduction de *Blobs* dans des programmes par objets. Ils ont observé que, souvent, les *Blobs* sont crées par les développeurs par accident par l'agrégation incrémentale de fonctionnalités à une classe jouant un rôle central dans un programme. Pourtant, dans certains autres cas, des *Blobs* sont crées intentionnellement par les développeurs pour résoudre des problèmes particuliers de conception ou d'implantation, par exemple quand une classe ne peut être décomposée pour des raisons de performance. Ils ont montré qu'il est possible de distinguer les *Blobs* créés intentionnellement des *Blobs* accidentels en étudiant l'évolution des classes *Blobs* dans le temps.

Khomh *et al.* [KHO 09a] ont étudié la propension aux changements de classes avec ou sans mauvaises odeurs. Ils ont identifié les classes ayant une ou plusieurs mauvaises odeurs parmi 29 dans neuf versions de Azureus et treize versions d'Eclipse. Ils ont montré que dans la plupart des versions des deux programmes, les classes avec des mauvaises odeurs sont plus sujettes à des changements que les autres et que certaines mauvaises odeurs sont plus corrélées aux changements que d'autres. Ces résultats confirment *a posteriori* les travaux précédents sur la spécification et la détection des mauvaises odeurs et des anti-patrons.

3.2.5. *Bilan*

Les nombreux modèles de qualité présentés dans la littérature tentent de définir les caractéristiques de qualité le plus clairement possible pour qu'elles soient mesurables objectivement mais sans avoir, à ce jour, atteint un consensus sur les caractéristiques de qualité les plus importantes et les relations entre ces caractéristiques entre-elles et entre les caractéristiques et les attributs internes des classes.

Ainsi, les travaux précédemment cités concernant les modèles de qualité définissent dix-sept caractéristiques de qualité, résumées par la table 3.1. Les travaux précédents portant sur les impacts des patrons de conception et anti-patrons considèrent sept caractéristiques de qualité, résumées par la table 3.2, ainsi que trois autre facteurs mois bien définis (conception et implantation) ou de plus bas niveau (métriques objets). Les deux tables montrent que tous les travaux précédents ne s'accordent pas sur les caractéristiques de qualité à étudier. Cependant, certaines caractéristiques semblent recevoir une certaine unanimité et sont les objets de nombreux travaux, en particulier la changeabilité et la propension aux fautes. C'est pourquoi notre modèle présenté dans la suite de ce chapitre considère aussi ces deux caractéristiques.

Caractéristiques	Références
Changeabilité	[MCC 77], [BOE 78]
Compréhensibilité	[BOE 78], [BAN 02]
Correction	[MCC 77]
Efficacité	[DRO 96], [MCC 77], [BOE 78], [ISO91], [BAN 02]
Extensibilité	[BAN 02]
Fiabilité	[DRO 96], [MCC 77], [BOE 78], [ISO91]
Flexibilité	[MCC 77], [BAN 02]
Fonctionnalité	[DRO 96], [ISO91], [BAN 02]
Intégrité	[MCC 77]
Interopérabilité	[MCC 77]
Maintenabilité	[DRO 96], [MCC 77], [BOE 78], [ISO91]
Maturité du processus	[DRO 96]
Portabilité	[DRO 96], [MCC 77], [BOE 78], [ISO91]
Réutilisabilité	[DRO 96], [MCC 77], [BAN 02]
Testabilité	[MCC 77], [BOE 78]
Utilité	[BOE 78]
Utilisabilité	[DRO 96], [MCC 77], [BOE 78], [ISO91]

Tableau 3.1. *Résumé des caractéristiques de qualité cité dans la littérature sur les modèles de qualité*

Caractéristiques	Références
Changeabilité	[BIE 03], [Di 08], [AVE 07], [OLB 09], [KHO 09a]
Compréhensibilité	[LAN 95], [WYD 98], [IGN 04], [BOI 06]
Flexibilité	[WYD 98]
Maintenabilité	[MCN 01], [BRO 98], [IGN 04]
Modularité	[WYD 98], [MCN 01]
Propension aux fautes	[VOK 04], [NG 07], [WEI 07], [ZIM 07]
Réutilisabilité	[WYD 98], [MCN 01]
Conception	[WEN 01]
Implantation	[WEN 01]
Métriques objets	[MAS 99], [MCN 01], [VOK 04], [ZIM 07]

Tableau 3.2. *Résumé des caractéristiques de qualité cité dans la littérature sur l'impact des patrons de conception et des anti-patrons sur la qualité*

Dans la suite de ce chapitre, nous considérons deux caractéristiques de qualité des classes : leurs propensions aux changements et aux fautes. Nous construisons sur les travaux précédents qui ont montré l'impact des anti-patrons et patrons de conception sur la qualité des programmes par objets et utilisons la participation de classes à des occurrences d'anti-patrons ou de patrons de conception, en plus de métriques de classes, pour construire un modèle de qualité prenant en compte la conception des programmes par objets.

3.3. DEQUALITE et PQMOOD

3.3.1. *DEQUALITE*

Nous proposons la méthode DEQUALITE (*Design Enhanced QUALITy Evaluation*) pour construire des modèles de qualité qui évaluent la propension aux changements et aux fautes des classes des programmes par objets en prenant en compte leur conception, en particulier par l'intermédiaire de la participation des classes à des patrons et anti-patrons de conception.

DEQUALITE est fondée sur l'observation que la conception d'un programme a un impact sur sa qualité ; par exemple quand l'implantation d'un motif de conception est reconnue dans une partie du programme, la compréhension par les développeurs des classes participantes est facilitée.

À partir de cette observation, nous proposons les cinq étapes de DEQUALITE, inspirées du travail de Dromey [DRO 95], pour construire des modèles de qualité qui lient des attributs internes et caractéristiques de qualité des programmes par objets à leur conception :

– identifier un ensemble de caractéristiques de qualité ;

– identifier un ensemble d'attributs internes capturant la conception ;

– choisir une technique de modélisation et de prédiction de la qualité ;

– construire un modèle liant attributs et caractéristiques de qualité ;

– évaluer le modèle et ses performances. Si les performances sont inacceptables, recommencer à l'étape 2 ci-dessus.

3.3.2. *Le modèle de qualité PQMOOD*

Nous appliquons maintenant étape par étape notre méthode DEQUALITE afin de construire le modèle de qualité PQMOOD.

3.3.2.1. *Étapes 1 et 2*

Nous choisissons les propensions aux changements et aux fautes des classes comme caractéristiques de qualité.

Nous mesurons la conception des classes à l'aide des deux variables discrètes :

– AP avec les valeurs disjointes : $no\ smell$, $one\ smell$ et $more\ smell$. Nous assignons une valeur à la variable discrète AP en identifiant et comptant le nombre d'anti-patrons auxquels chaque classe fait partie avec notre méthode DECOR [MOH 10] : si une classe n'est affectée par aucun anti-patron, $AP = no\ smell$, par un anti-patron, $AP = one\ smell$ et par deux ou plus, $AP = more\ smell$;

– DP avec les valeurs disjointes : $no\ role$, $one\ role$ et $more\ role$. Nous assignons aussi une valeur à la variable discrète DP en identifiant et comptant le nombre de patrons auxquels chaque classe fait partie avec notre méthode DeMIMA [GUÉ 08].

Les anti-patrons se manifestent dans le code source sous la forme de mauvaises odeurs [FOW 99] et les patrons sous la forme de classes jouant certains rôles dans les motifs proposés par les patrons [GUÉ 08]. Aussi, nous utilisons les valeurs discrètes $no\ smell$, $one\ smell$ et $more\ smell$ pour attribuer une étiquette à chaque classe d'un programme par objets par rapport à la mesure AP (respectivement $no\ role$, $one\ role$ et $more\ role$ pour DP).

Nous utilisons aussi les attributs internes de qualité suivants dont la pertinence a été montrée dans les travaux de Zimmermann *et al.* [ZIM 07], ainsi que leurs mesures respectives :

– pre : le nombre de fautes prédistribution dans les classes, six mois avant la date de la version du programme dont la qualité est évaluée ;

– $post$: le nombre de fautes postdistribution dans les classes, six mois après la date de mise à disposition du programme dont la qualité est évaluée ;

– VG_{sum} : la complexité de McCabe accumulée de chaque classe ;

– $TLOC$: le nombre total de lignes de code dans chaque classe, c'est à dire incluant les champs, les constructeurs statiques ou non ;

– $MLOC_{sum}$: la somme du nombre de lignes de code des méthodes d'une classe.

3.3.2.2. *Étape 3*

Nous utilisons la technique des réseaux bayésiens (BBN pour *Bayesian Belief Network*). Cette technique nous permet d'obtenir un modèle de qualité plus adaptable et acceptant une plus grande variation dans ses données d'entrées que les modèles obtenus dans la littérature par régression logistique, par exemple Zimmermann *et al.* [ZIM 07]. En effet, un développeur peut intégrer son expertise ainsi que son contexte d'évaluation de la qualité dans un réseau bayésien en modifiant les relations entre nœuds d'entrée et de sortie et en changeant les tables de probabilités des nœuds.

Un BBN est un graphe orienté et acyclique [PEA 88]. Dans ce graphe, chaque variable aléatoire X_i, représentant une caractéristique d'une classe, est dénotée par un nœud. Un arc orienté entre deux nœuds indique une dépendance probabiliste entre la variable dénotée par le nœud parent vers celle du nœud enfant. Ainsi, le réseau dénote l'hypothèse que les valeurs de chaque nœud X_i dans celui-ci sont dépendantes conditionnellement sur les valeurs de leurs parents. Chaque nœud X_i dans le réseau est associé à une table de probabilités conditionnelles qui spécifie la distribution des probabilités de toutes ses valeurs possibles, pour chaque combinaison des valeurs

des parents. Ainsi, un BBN est une fonction de classification $f: R^d \mapsto C$ qui associe une étiquette dans un ensemble de classes $C = \{c_1, \ldots, c_q\}$ aux observations $\mathbf{x} = (a_1, \ldots, a_d) \in R^d$, valeurs des X_i.

Un développeur a besoin de deux informations pour construire un BBN : la structure du réseau, sous la forme de nœuds et d'arcs (relations de causes à effets) et les tables de probabilités conditionnelles décrivant le processus de décision entre chaque nœud. En structurant le réseau, le développeur assure que le processus de décision est valide. Les probabilités conditionnelles peuvent être apprises par entraînement du réseau sur des données historiques ou entrées directement par le développeur qui construit le modèle. Des tables de probabilités conditionnelles appropriées assurent que le BBN est bien calibré et quantitativement valide.

3.3.2.3. *Étape 4*

Nous construisons maintenant un modèle de qualité liant les caractéristiques de qualité sélectionnées à l'étape 1, c'est-à-dire la propension aux changements et aux fautes de classes aux attributs internes de l'étape 2, c'est-à-dire la conception des classes mesurée par AP et DP. La structure du BBN est comme suit : les nœuds d'entrée qui sont des variables correspondant aux probabilités $P(AP)$ et $P(DP)$ que les classes d'un programme par objets aient aucune, une ou plusieurs mauvaises odeurs et jouent aucun, un ou plusieurs rôles dans un motif de conception. Nous calculons ces probabilités en utilisant les fréquences d'apparition de mauvaises odeurs dans les classes du programme ainsi que les fréquences de participation de ces classes dans des motifs de conception. Par exemple, la probabilité :

$$P(AP = one\ smell) = \frac{\text{nombre de classes affectées par un seul anti} - \text{patron}}{\text{nombre total de classes du programme}}$$

Les nœuds de sortie X_{i_s} décrivent les probabilités qu'une classe soit sujette à des changements ou fautes à partir d'une observation $(a_1, ..., a_d)$, c'est-à-dire un vecteur de valeurs d'entrée décrivant les nombres de patrons et d'anti-patrons de conception dans lesquels la classe participe. Par exemple, une valeur a_i peut correspondre à la mesure AP, indiquant que la classe participe dans un nombre d'anti-patrons donné.

Nous utilisons la probabilité des nœuds de sortie et un seuil t pour classifier les classes d'un programme. Par exemple, dans le cas de la propension aux changements (respectivement fautes), une classe d'un programme par objet sera classifiée dans $C = \{change - prone, not\ change - prone\}$ (respectivement, $F = \{fault - prone, not\ fault - prone\}$) et considérée comme $change - prone$ si $X_{i_s} > t$ et comme $not\ change - prone$ si $X_{i_s} \leq t$. Dans le reste de ce chapitre, nous choisissons $t = 0,5$ comme première approximation pour simplement séparer en deux groupes les classes d'un programme par objets. Nous utilisons à la fois la classification des classes dans les ensembles C et F et leur ordonnancement suivant leurs probabilités d'être sujettes à des changements et à des fautes.

Un BBN permet de gérer des cooccurrences de patrons et d'anti-patrons de conception qui créeraient une dépendance entre les variables AP et DP car, comme le remarque Zhang [ZHA 04], les BBN sont robustes aux interdépendances entre nœuds d'un même niveau.

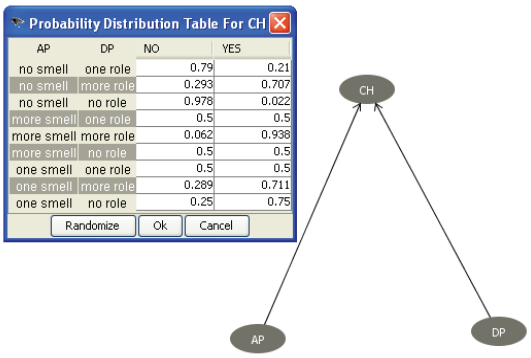


Figure 3.1. *Sous-ensemble du BBN de PQMOOD pour la propension des classes aux changements*

La figure 3.1 présente la structure simplifiée d'un BBN pour la propension aux changements. Nous l'utilisons dans la suite comme exemple pour expliquer notre modèle de qualité à base de BBN, PQMOOD.

Nœuds d'entrée. Les nœuds d'entrée du BBN sur la figure 3.1 sont $P(AP)$ et $P(DP)$. Pour chaque nœud, nous calculons les distributions des probabilités en utilisant des données historiques comme suit :

– pour $P(AP)$, nous utilisons les fréquences des classes participant à aucune, une, ou plusieurs mauvaises odeurs afin de calculer la probabilité qu'une classe c_i participe respectivement dans aucun, un ou plusieurs anti-patrons (c'est-à-dire $P(AP)$) ;

– de façon similaire, nous utilisons les fréquences des classes jouant aucun, un ou plusieurs rôles dans un motif de conception afin de calculer $P(DP)$.

Nœuds de sortie. La probabilité d'un nœud de sortie, par exemple la propension aux changements, est calculée à partir des probabilités des nœuds d'entrée ($P(AP)$ et $P(DP)$) en utilisant le théorème de Bayes. Chaque nœud de sortie a une table de probabilités conditionnelles qui décrit la probabilité d'étiqueter une classe comme, par exemple $change - prone$ ou $not\ change - prone$, en fonction des probabilités d'entrée $P(AP)$ et $P(DP)$ et d'un seuil de décision t. Pour la propension aux changements dans le BBN de la figure 3.1, la probabilité d'une classe d'être sujette à des changements se calcule grâce à la table 3.3 de probabilités conditionnelles.

$P(CH\|no\ smell\ \&\ no\ role)$
$P(CH\|no\ smell\ \&\ one\ role)$
$P(CH\|no\ smell\ \&\ more\ role)$
$P(CH\|one\ smell\ \&\ no\ role)$
$P(CH\|one\ smell\ \&\ one\ role)$
$P(CH\|one\ smell\ \&\ more\ role)$
$P(CH\|more\ smell\ \&\ no\ role)$
$P(CH\|more\ smell\ \&\ one\ role)$
$P(CH\|more\ smell\ \&\ more\ role)$

Tableau 3.3. *Exemple de table de probabilités conditionnelles*

3.3.3. *Validation de PQMOOD*

Suivant l'étape 5 de DEQUALITE, nous réalisons maintenant des expériences pour valider le modèle de qualité à base de BBN, PQMOOD.

Le but de nos validations est de montrer qu'un BBN utilisant des informations de conception peut prédire la propension aux changements et aux fautes des classes de programmes par objets. Le focus qualité de nos validations est de fournir un ensemble ordonné de classes sujettes aux changements et aux fautes qui met en avant les classes qui ont les plus grandes probabilités de changer ou d'avoir des fautes. Notre perspective est que les développeurs, qui évaluent la qualité des classes et sont intéressés à identifier les classes les plus problématiques, ont besoin d'identifier ces classes avec le moins d'effort possible. Le contexte est celui du développement et de la maintenance.

Nous cherchons à répondre à deux questions de recherche :

– QR1 : avec quelle précision et quel rappel le modèle PQMOOD peut-il évaluer la propension aux changements et aux fautes des classes d'un programme ?

– QR2 : est-ce que les résultats de notre modèle de qualité incluant la conception sont meilleurs que les résultats sans prendre en compte la conception des classes ?

Pour répondre à QR1, nous étudions l'efficacité de notre BBN dans deux scénarios. D'abord, nous faisons l'hypothèse que des données historiques sont disponibles pour une version k d'un programme par objets, c'est-à-dire des données sur les changements et les fautes dans les classes du programme. Nous utilisons ces données pour calibrer le BBN construit à l'étape 4. Nous appliquons ensuite ce BBN calibré sur la version suivante $(k + 1)$ du même programme pour prédire les changements et fautes dans les classes et les comparer avec la réalité. Ensuite, nous utilisons des données hétérogènes : nous calibrons le BBN avec les données sur les changements d'un programme et l'appliquons sur un programme différent.

Pour répondre à QR2, nous réalisons d'abord une régression logistique pas-à-pas sur les mesures identifiées aux étapes 1 et 2 pour ne garder que les plus pertinentes dans l'évaluation de la propension aux changements des classes. Ensuite, nous construisons deux nouveaux modèles de qualité supplémentaires en suivant DEQUALITE : un modèle qui n'utilise que les valeurs des métriques de classes retenues par la régression logistique pas-à-pas et un autre qui combine ces métriques de classes et les mesures de conception.

Nous utilisons les trois programmes à code source libre suivants dans notre validation : Rhino, Mylyn et Eclipse-JDT. Eclipse-JDT est l'environnement de développement pour Java d'Eclipse ; Mylyn est un gestionnaire d'activités pour Eclipse ; Rhino est un interpréteur ECMA/Javascript en Java. Le tableau 3.4 présente les principales caractéristiques de ces programmes.

Programmes	Nombres de		
	Versions	Classes	LOCs
Eclipse-JDT	1.0	1,382	257,605
Mylyn	2.0.0	1,759	185,169
Rhino	1.4R3	89	30,748

Tableau 3.4. *Caractéristiques des programmes*

3.3.3.1. *QR1 : précisions et rappels de PQMOOD*

Scénario 1 : validation intraprogramme. Dans ce premier scénario, nous étudions comment l'histoire des changements des classes d'un programme par objets peut être utilisée pour prédire la propension aux changements des classes dans de futures versions du programme. Nous entraînons notre modèle sur la première version d'un programme et l'utilisons pour prédire la propension aux changements des classes dans les versions suivantes, tel qu'expliqué à l'étape 4 de DEQUALITE.

La figure 3.2 présente les résultats sur Rhino où le modèle est entraîné sur la version 1.4R3 et appliqué sur la version 1.5R1. Les classes du programme sont ordonnées en fonction de leur probabilité d'être sujettes à des changements : de la plus grande probabilité, à la plus petite. De cet ensemble ordonné, nous construisons des ensembles de classes $suggested$, qui incluent les x premières classes, $1 < x <$ nombre total de classes du programme. Chaque sous-figure montre la précision et le rappel de ces différents ensembles de classes, calculés de la manière suivante :

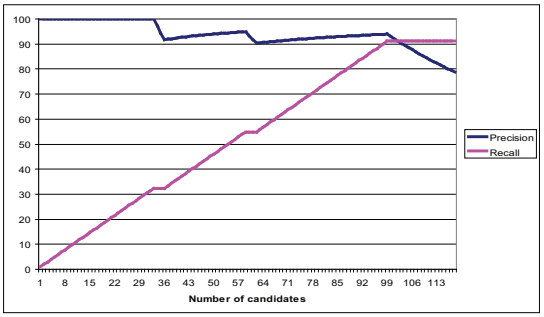
$$precision = \frac{|correct \cap suggested|}{|suggested|}$$

$$rappel = \frac{|correct \cap suggested|}{|correct|}$$

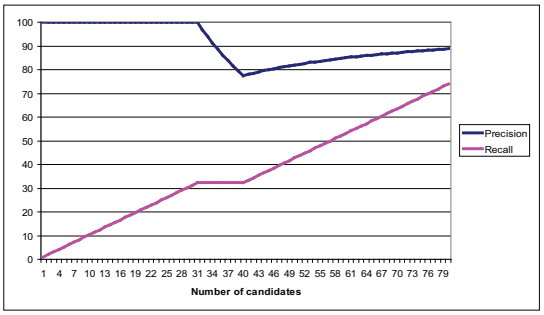
où $correct$ représente l'ensemble connu des classes ayant changé et $suggested$ l'ensemble des classes choisies.

Généralement, nous obtenons une précision moyenne supérieure à 80 % pour prédire la propension aux changements dans les futures versions de Rhino. De plus, la figure 3.2 montre que notre modèle obtient en moyenne 100 % de précision sur les 33 premières classes (les 28 % des classes qui sont les plus probables de changer).

Nous répliquons cette expérience sur Mylyn pour les versions 2.0.0 et 2.1 et obtenons 100 % de précision pour les 95 première classes (29 % des classes candidates). Nous confirmons ainsi l'efficacité de notre modèle de qualité à prédire les classes sujettes à des changements.



a) Changements : Rhino



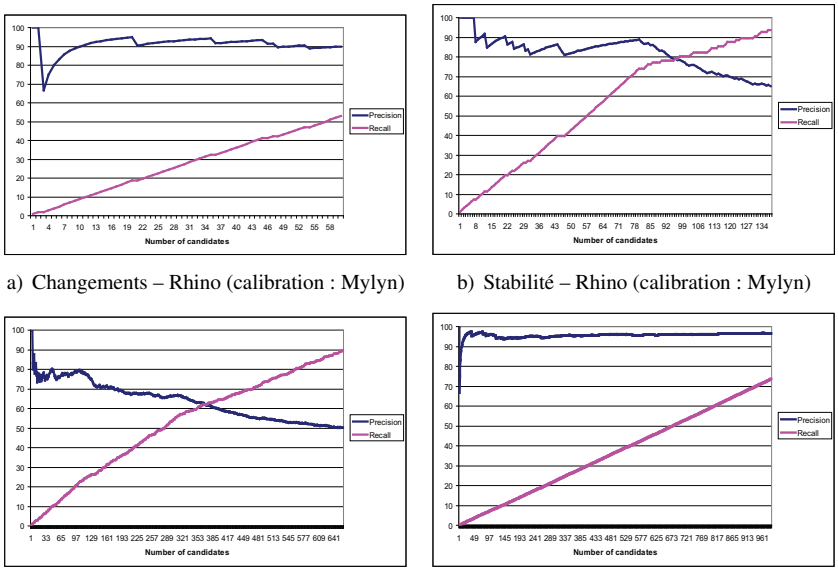
b) Stabilité : Rhino

Figure 3.2. *Calibration intraprogramme : précision et rappel*

Ayant obtenu ces taux élevés de précisions ($> 80\%$) et de rappels ($> 90\%$) dans le scénario intraprogramme, nous étudions maintenant l'utilisation de notre modèle pour prédire d'un programme à un autre les classes sujettes aux changements.

Scénario 2 : validation interprogrammes. Dans ce second scénario, nous faisons l'hypothèse qu'un développeur a accès aux données historiques d'un programme et qu'il voudrait utiliser ces données pour calibrer son modèle de qualité et ensuite appliquer ce modèle sur les classes d'un autre programme.

La figure 3.3 montre que notre modèle calibré sur Rhino obtient une précision supérieure à 50 % pour prédire les classes sujettes à des changements dans Mylyn et à 90 % dans Eclipse-JDT. De même, notre modèle entraîné sur Mylyn obtient une précision supérieure à 90 % pour Rhino. Le rappel moyen est de 75 %.



a) Changements – Rhino (calibration : Mylyn)

b) Stabilité – Rhino (calibration : Mylyn)

c) Changements – Mylyn (calibration : Rhino)

d) Stabilité – Mylyn (calibration : Rhino)

Figure 3.3. *Validation interprogrammes*

Ces résultats suggèrent que même en l'absence de données historiques pour un programme, un développeur peut utiliser un modèle calibré sur un autre programme et obtenir des précisions et des rappels acceptables. Ces résultats montrent aussi qu'un BBN peut être construit avec des données extérieures à une entreprise et adapté à son contexte avec succès.

3.3.3.2. *QR2 : PQMOOD et différentes mesures d'entrée*

Nous répondons maintenant à notre deuxième question de recherche portant sur le choix des mesures d'entrée du BBN pour prédire les propensions des classes aux changements et aux fautes.

Nous confirmons d'abord l'importance des caractéristiques de qualité portant sur la conception et de leurs mesures en réalisant une régression logistique pas à pas. La régression logistique pas à pas passe au crible un ensemble de variables indépendantes pour ne retenir que celles qui sont importantes pour décrire la variable dépendante [SHE 11]. Parmi les six variables indépendantes définies à l'étape 2 ci-dessus, la régression logistique pas à pas retient pre, $MLOC_{sum}$, $TLOC$, AP et DP ; confirmant ainsi l'importance de la conception (AP et DP) dans la prédiction des classes sujettes aux fautes. Le tableau 3.5 présente les coefficients de cette régression logistique.

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	-3.4222	0.3160	-10.83	**0.0000**
pre	0.2021	0.0325	6.23	**0.0000**
$MLOC\_sum$	0.0087	0.0032	2.72	**0.0065**
$TLOC$	-0.0042	0.0026	-1.59	0.1120
VG_sum	0.0025	0.0053	0.47	0.6366
$AP = one\ smell$	0.8158	0.3466	2.35	**0.0186**
$AP = more\ smell$	1.5209	0.2872	5.29	**0.0000**
$DP = one\ role$	-0.3534	0.3331	-1.06	0.2887
$DP = more\ role$	0.4966	0.1705	2.91	**0.0036**

Tableau 3.5. *Coefficients de la régression logistique*

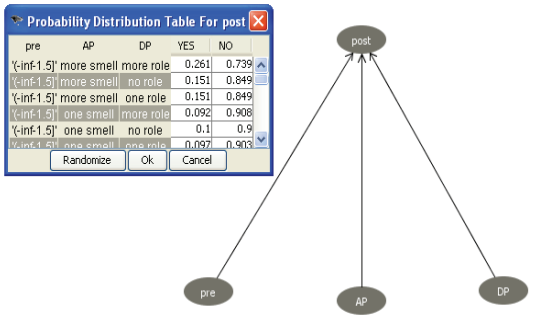
En conséquence, nous concluons que la conception des programmes fournit une information complémentaire aux métriques et est importante pour prédire les classes sujettes aux fautes.

Suivant les résultats de la régression logistique pas à pas, nous créons deux modèles de qualité additionnels basés sur les BBN avec (1) les métriques de classes pre, $MLOC_{sum}$ et $TLOC$ seules et (2) une combinaison des métriques de classes et de conception. La figure 3.4b présente la structure du modèle de qualité avec toutes les métriques. En structurant le BBN à partir des résultats de la régression logistique pas à pas, nous assurons que le processus de décision est amélioré. Nous entraînons ces modèles et les appliquons sur les versions 2.0 et 2.1 d'Eclipse-JDT et obtenons les performances résumées dans le tableau 3.6 en fonction de trois ensembles de

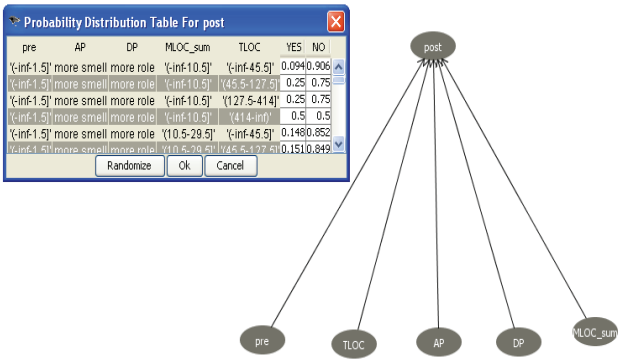
– métriques $= \{pre, VG_{sum}, TLOC, MLOC_{sum}\}$: nous utilisons les mêmes métriques que celles utilisées par Zimmermann *et al.* [ZIM 07] ;

– conception $= \{AP, DP\}$: nous construisons un modèle à partir des seules données de conception des classes, c'est-à-dire les nombres de rôles joués par les classes dans les patrons (DP) et leurs participations aux anti-patrons de conception (AP) ;

– conception + métriques ($=$ métriques $\cup$ conception) : nous construisons un modèle qui utilise les métriques de Zimmermann *et al.* [ZIM 07] et les données sur la conception des classes.



a) Conception



b) Combinaison de la conception et des métriques

Figure 3.4. *Modèles de qualité pour la propension aux fautes*

Entraînement	Test	Métriques		Conception		Conception + Métriques	
		Précision	Rappel	Précision	Rappel	Précision	Rappel
2.0	2.0	0.63	0.25	0.61	0.13	**0.63**	**0.27**
	2.1	0.40	0.30	**0.65**	0.15	0.41	**0.31**
2.1	2.0	0.62	0.22	0.54	0.18	**0.62**	**0.22**
	2.1	0.49	0.24	**0.65**	0.14	0.50	**0.24**

Tableau 3.6. *Modèles de qualité : précision et rappel sur Eclipse-JDT*

Dans notre étude, le modèle combinant des métriques de classes et de conception surpasse les modèles construits uniquement avec des données sur les classes ou sur leur conception. Ce résultat est cohérent avec les résultats obtenus par régression logistique et présentés dans le tableau 3.7. De plus, dans le cas de Eclipse 2.1 les BBN contenant exclusivement des données sur la conception des classes surpassent ceux

construits avec les métriques ou avec une combinaison de la conception et des métriques. Ce résultat renforce l'importance de la conception des programmes pour la prédiction des fautes.

3.3.4. *Discussions*

Nous discutons maintenant nos expériences et l'utilisation de DEQUALITE par des développeurs.

3.3.4.1. *Utilisation des BBN dans un contexte industriel*

Nos modèles ordonnent les classes sujettes aux changements et aux fautes qui devraient donc être inspectées par des développeurs, nous avons montré dans un autre travail [KHO 11] que cet ordonnancement est plus avantageux qu'un ordre aléatoire pour un développeur qui veut identifier les classes les plus problématiques.

Dans la section 3.3.3.1, nous avons montré qu'un modèle construit avec DEQUALITE et calibré avec des données externes, (c'est-à-dire des données provenant d'un programme différent de celui sur lequel le modèle est appliqué) peut identifier avec précision des classes sujettes aux changements. Les programmes Rhino et Mylyn étaient de natures différentes (un interpréteur JavaScript et une extension à Eclipse) et développés par des équipes différentes. Ainsi, un développeur pourrait utiliser des données publiques disponibles par exemple dans des programmes à code source libre pour construire et calibrer un modèle de qualité applicable dans son contexte industriel pour prédire les classes qui ont une plus grande propension à changer.

Les modèles de qualité construits en suivant DEQUALITE et en utilisant des données de conception et des valeurs de métriques sont meilleurs que les modèles utilisant seulement des métriques. De plus, ils indiquent aux développeurs les problèmes précis des classes puisqu'ils utilisent des données sur la participation des classes dans des anti-patrons et patrons de conception, qui définissent des styles particuliers de conception ; alors que les modèles basés seulement sur des métriques ne fournissent qu'un score sans contexte et sans indication des raisons possibles pour qu'une classe soit plus ou moins sujette à des changements ou à des fautes.

De plus, comme discuté dans la section 3.3.3.2, un modèle de qualité construit avec DEQUALITE peut être spécialisé et amélioré avec d'autres types de données que les développeurs jugent pertinentes dans leur contexte industriel. La régression logistique pas à pas fournit un outil intéressant pour améliorer les modèles. En plus d'être dirigée par les données, elle permet d'explorer des modèles différents en construisant une collection de modèles qui n'aurait sinon pas pu être étudiée[8] [SHT 01].

---

8. « [Logistic regression] allows for the examination of a collection of models which might not otherwise have been examined ».

Entraînement	Test	Métriques		Conception		Métrique + Conception	
		Précision	Rappel	Précision	Rappel	Précision	Rappel
2.0	2.0	0.68	0.22	0.63	0.12	**0.71**	0.24
	2.1	0.42	0.25	**0.65**	0.13	**0.44**	0.26
2.1	2.0	0.60	0.11	0.58	0.13	**0.62**	0.12
	2.1	0.61	0.16	**0.64**	0.14	**0.62**	0.17

Tableau 3.7. *Régression logistique : précision et rappel*

3.3.4.2. *PQMOOD et le modèle de Zimmermann* et al.

Zimmermann *et al.* [ZIM 07] ont mis en correspondance, d'une part, les fautes répertoriées dans les rapports de bogues des version 2.0 et 2.1 d'Eclipse et, d'autre part, les classes de ces versions. Ils rapportent ensuite les résultats de la construction d'un modèle de qualité pour évaluer la propension aux fautes des classes à l'aide de régression logistique.

Nous réutilisons les données de l'étude de Zimmermann *et al.*, qu'ils ont rendues publiques, et construisons trois modèles de qualité par régressions logistiques en utilisant les ensembles de variables précédemment définis, *i.e.*, Métriques, Conception et Conception + Métriques :

– métriques $= \{pre, VG_{sum}, TLOC, MLOC_{sum}\}$: avec cet ensemble, nous répliquons le modèle de régression logistique de Zimmermann *et al.* à des fins de comparaison [ZIM 07]. Nous appelons ce modèle $RL_{Metrics}$;

– conception $= \{AP, DP\}$: avec cet ensemble, nous construisons un modèle à partir des seules données de conception des classes, c'est-à-dire les nombres de rôles joués par les classes dans les patrons (DP) et leurs participations aux anti-patrons de conception (AP). Nous appelons ce modèle RL_{Design} ;

– conception + métriques $=$ métriques $\cup$ conception : avec cet ensemble, nous construisons un modèle qui utilise les métriques de Zimmermann *et al.* [ZIM 07] et les données sur la conception des classes. Nous appelons ce modèle $RL_{Metrics \cup Design}$.

Nous répliquons l'étude de Zimmermann *et al.* [ZIM 07] et comparons les précisions et les rappels des trois modèles $RL_{Metrics}$, RL_{Design} et $RL_{Metrics \cup Design}$.

Le tableau 3.7 rapporte les résultats obtenus par les trois modèles. Les résultats du modèle $RL_{Metrics \cup Design}$ (en gras) sont toujours meilleurs que ceux des deux autres modèles, renforçant à nouveau notre observation que la conception d'un programme a un impact sur sa qualité. Nous montrons ainsi qu'un modèle prenant en compte la conception des programmes à une meilleure précision et un meilleur rappel pour la prédiction des fautes qu'un modèle utilisant seulement des métriques.

De plus, comme présenté dans le tableau 3.7, les données sur la conception des programmes peuvent, dans certains contextes, être plus importantes que les métriques : pour Eclipse-JDT 2.1, les modèles construits avec des données sur la conception des programmes ont des résultats en général meilleurs que le modèle sans.

Nous expliquons l'exception de Eclipse-JDT 2.1 par le fait que la régression logistique pas-à-pas, étant une méthode dirigée par les données, ne permet pas nécessairement d'obtenir le « meilleur » modèle en général, car elle cherche à adapter le modèle qu'elle construit aux données existantes avec le danger de suradapter (*overfit*) celui-ci au bruit dans les données fournies [SHT 01]. Pour la version 2.1 d'Eclipse-JDT, le meilleur modèle est celui utilisant seulement les données de conception.

3.3.4.3. *PQMOOD et QMOOD*

Le modèle hiérarchique QMOOD, introduit par Bansiya et Davis [BAN 02], est le plus récent des modèles de qualité publiés et le plus utilisé dans la littérature. Le succès de ce modèle peut être expliqué par sa validation sur des programmes de tailles moyennes et grandes. QMOOD définit six équations pour évaluer : réutilisabilité, flexibilité, compréhension, fonctionnalité, extensibilité et efficacité.

Pour comparer notre modèle de qualité PQMOOD à QMOOD, nous étudions si PQMOOD retourne le même ensemble de classes problématiques que QMOOD, bien que les deux modèles mesurent des attributs et caractéristiques de qualité différents. Nous implantons les six équations de QMOOD décrites dans [BAN 02] et calibrons le BBN présenté sur la figure 3.1 sur Rhino 1.4R3. Nous appliquons alors les deux modèles sur Mylyn 2.0.0 et observons que, parmi les 20 % premières classes considérées comme les moins réutilisables, flexibles et extensibles par QMOOD :

– 71 % sont sujettes à des changements ;

– 98 % sont prédites comme sujettes à des changements ;

– 69 % sont dans l'ensemble des vingt premiers pourcents des classes.

En conséquence, même si notre modèle de qualité n'a pas pour objectif de mesurer les mêmes attributs et caractéristiques que QMOOD et qu'il est calibré sur un autre programme que QMOOD (Rhino), les résultats montrent qu'il peut être aussi précis que QMOOD pour détecter des classes problématiques dans un programme.

De plus, avec PQMOOD, un développeur pourrait calibrer le modèle pour prédire les mêmes caractéristiques de qualité que QMOOD si les données appropriées sont disponibles, par exemple classes qui sont moins flexibles que les autres.

3.4. Conclusion

Dans ce chapitre, nous avons présenté un l'état de l'art sur les modèles de qualité et introduit notre méthode DEQUALITE et son instantiation, PQMOOD, pour la mesure

de la propension aux changements et aux fautes des classes. Ce chapitre surmonte trois défis liés à la création de modèles de qualité :

– le choix des caractéristiques de qualité à évaluer ;

– la création et la paramétrisation du modèle ;

– la validation du modèle obtenu.

Nous avons répondu à ces défis par un état de l'art sur la qualité et les anti-patrons et patrons de conception, une méthode de construction de modèles de qualité et une validation pour répondre aux questions de recherche :

– QR1 : avec quelle précision et quel rappel notre modèle peut-il évaluer la propension aux changements et aux fautes des classes d'un programme ?

– QR2 : est-ce que les résultats de notre modèle de qualité incluant la conception sont meilleurs que les résultats sans prendre en compte la conception des classes ?

Les résultats de notre travail sont une méthode, DEQUALITE, et son instantiation, PQMOOD, pour la mesure de la propension aux changements et aux fautes des classes. Nous avons choisi la propension aux changements et aux fautes à partir de la littérature, en considérant que de nombreux travaux s'intéressent à ces caractéristiques de qualité. Nous avons basé notre méthode sur la méthode de Dromey [DRO 96], reprise également par Bansiya et Davis [BAN 02] pour leur modèle QMOOD. Nous avons construit PQMOOD avec des réseaux bayésiens et obtenu un modèle opérationnel en utilisant plusieurs versions de différents programmes comme oracle : Eclipse-JDT, Mylyn et Rhino.

Nous avons finalement répondu aux deux questions de recherche QR1 et QR2. Les réponses à ces questions sont positives en général et en particulier par comparaison avec deux modèles de qualité existants : ceux de Zimmermann *et al.* [ZIM 07] et de Bansiya et Davis [BAN 02]. De plus, nos validations montrent que PQMOOD peut être entraîné sur un programme et appliqué sur un autre programme différent avec de bonnes précisions et de bons rappels. Ainsi, un développeur peut bénéficier d'un modèle de qualité construit sur d'autres programmes de la même entreprise ou même d'entreprises différentes.

Nos travaux futurs incluent l'utilisation de PQMOOD pour étudier l'évolution de la qualité des programmes et des restructurations opérées par les développeurs. Nous allons aussi reproduire nos évaluations sur d'autres programmes et développer de nouveaux modèles de qualité, en suivant notre méthode DEQUALITE, qui prennent en compte d'autres attributs et caractéristiques de qualité.

Remerciements. Ce travail a été en parti financer par la chaire de recherche du CRSNG sur les patrons logiciels et les patrons de logiciels.

3.5. Bibliographie

[ABB 11] ABBES M., KHOMH F., GUÉHÉNEUC Y.-G., ANTONIOL G., "An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, On Program Comprehension", KANELLOPOULOS Y., MENS T., Eds., *Proceedings of the* 15^{th} *European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE Computer Society Press, March 2011, Best paper. 10 pages.

[AVE 07] AVERSANO L., CANFORA G., CERULO L., GROSSO C. D., DI PENTA M., "An Empirical Study on the Evolution of Design Patterns", BERTOLINO A., Ed., *Proceedings of the* 6^{th} *joint meeting of the European Software Engineering Conference and the* 14^{th} *Symposium on Foundations of Software Engineering*, ACM Press, pp. 385–394, September 2007.

[BAN 02] BANSIYA J., DAVIS C. G., "A Hierarchical Model for Object-Oriented Design Quality Assessment", *Transactions on Software Engineering*, vol. 28, n°1, pp. 4–17, IEEE Computer Society Press, January 2002.

[BEC 94] BECK K., JOHNSON R. E., "Patterns Generate Architectures", TOKORO M., PARESCHI R., Eds., *Proceedings of* 8^{th} *European Conference for Object-Oriented Programming*, Springer-Verlag, pp. 139–149, July 1994.

[BIE 01] BIEMAN J. M., ALEXANDER R., MUNGER III P. W., MEUNIER E., "Software Design Quality: Style and Substance", *Proceedings of the* 4^{th} *Workshop on Software Quality*, ACM Press, March 2001.

[BIE 03] BIEMAN J., STRAW G., WANG H., MUNGER III P. W., ALEXANDER R. T., "Design Patterns and Change Proneness: An Examination of Five Evolving Systems", BERRY M., HARRISON W., Eds., *Proceedings of the* 9^{th} *international Software Metrics Symposium*, IEEE Computer Society Press, pp. 40–49, September 2003.

[BOE 76] BOEHM B. W., BROWN J. R., LIPOW M., "Quantitative Evaluation of Software Quality", *Proceedings of the* 2^{nd} *international conference on Software engineering*, IEEE CS Press, pp. 592–605, 1976.

[BOE 78] BOEHM B. W., BROWN J. R., KASPAR H., LIPOW M., MCLEOD G., MERRITT M., *Characteristics of Software Quality*, Elsevier Science Ltd, 1st édition, june 1978.

[BOI 06] BOIS B. D., DEMEYER S., VERELST J., MENS T., TEMMERMAN M., "Does God Class Decomposition Affect Comprehensibility?", *Proceedings of the IASTED International Conference on Software Engineering*, IASTED/ACTA Press, pp. 346–355, February 2006.

[BRI 02a] BRIAND L. C., MELO W. L., WÜST J., "Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects", *Transactions on Software Engineering*, vol. 28, n°7, pp. 706–720, July 2002.

[BRI 02b] BRIAND L. C., WÜST J., "Empirical Studies of Quality Models in Object-Oriented Systems", *Advances in Computers*, vol. 56, Academic Press, 2002.

[BRO 98] BROWN W. J., MALVEAU R. C., BROWN W. H., MCCORMICK III H. W., MOWBRAY T. J., *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley and Sons, 1^{st} édition, March 1998.

[COP 05] COPLIEN J. O., HARRISON N. B., *Organizational Patterns of Agile Software Development*, Prentice-Hall, 1^{st} édition, 2005.

[DEI 09] DEISSENBOECK F., JUERGENS E., LOCHMANN K., WAGNER S., "Software Quality Models: Purposes, Usage Scenarios and Requirements", *Proceedings of the ICSE Workshop on Software Quality*, IEEE CS Press, pp. 9–14, May 2009.

[DEU 88] DEUTSCH M., WILLIS R., *Software Quality Engineering: A Total Technical and Management Approach*, Prentice Hall, February 1988.

[Di 08] DI PENTA M., CERULO L., GUÉHÉNEUC Y.-G., ANTONIOL G., "An Empirical Study of the Relationships between Design Pattern Roles and Class Change Proneness", MEI H., WONG K., Eds., *Proceedings of the 24^{th} International Conference on Software Maintenance (ICSM)*, IEEE Computer Society Press, September–October 2008, 10 pages.

[DRO 95] DROMEY R. G., "A Model for Software Product Quality", *Transactions on Software Engineering*, vol. 21, n°2, pp. 146–162, IEEE CS Press, February 1995.

[DRO 96] DROMEY R. G., "Cornering the Chimera", *IEEE Software*, vol. 13, n°1, pp. 33–43, IEEE CS Press, January 1996.

[EDE 03] EDEN A. H., KAZMAN R., "Architecture, Design, Implementation", DILLON L., TICHY W., Eds., *Proceedings of the 25^{th} International Conference on Software Engineering*, ACM Press, pp. 149–159, May 2003.

[EVA 87] EVANS M. W., MARCINIAK J. J., *Software Quality Assurance and Management*, Wiley, 1987.

[FOW 99] FOWLER M., *Refactoring – Improving the Design of Existing Code*, Addison-Wesley, 1^{st} édition, June 1999.

[GAM 94] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1^{st} édition, 1994.

[GUÉ 08] GUÉHÉNEUC Y.-G., ANTONIOL G., "DeMIMA: A Multi-layered Framework for Design Pattern Identification", *Transactions on Software Engineering (TSE)*, vol. 34, n°5, pp. 667–684, IEEE Computer Society Press, September 2008, 18 pages.

[HAN 02] HANNEMANN J., KICZALES G., "Design Pattern Implementation in Java and AspectJ", MATSUOKA S., Ed., *Proceedings of the 17^{th} Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM Press, pp. 161–173, November 2002.

[IEE 98] IEEE, "IEEE Std 1061-1998", *IEEE Standard for a Software Quality Metrics Methodology*, 1998.

[IGN 03] IGNATIOS D., IOANNIS S., LEFTERIS A., MANOS R., MARTIN S., "A controlled experiment investigation of an object oriented design heuristic for maintainability", *Journal of Systems and Software*, vol. 65, n°2, Elsevier, February 2003.

[IGN 04] IGNATIOS D., MARTIN S., MANOS R., IOANNIS S., "An empirical investigation of an object-oriented design heuristic for maintainability", *Journal of Systems and Software*, vol. 72, n°2, Elsevier, 2004.

[ISO 05] ISO/IEC, ISO/IEC 25000 - Software engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE, Rapport, 2005.

[ISO91] ISO/IEC, Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their Use, December 1991, ISO/IEC 9126:1991(E).

[JEA 09] JEANMART S., GUÉHÉNEUC Y.-G., SAHRAOUI H., HABRA N., "Impact of the Visitor Pattern on Program Comprehension and Maintenance", MILLER J., SELBY R., Eds., *Proceedings of the* 3^{rd} *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, IEEE Computer Society Press, October 2009, 10 pages.

[KAM 07] KAMPFFMEYER H., ZSCHALER S., "Finding the Pattern You Need: The Design Pattern Intent Ontology", ENGELS G., OPDYKE B., SCHMIDT D. C., WEIL F., Eds., *Proceedings of the* 10^{th} *International Conference on Model Driven Engineering Languages and Systems*, Springer, pp. 211–225, September–October 2007.

[KER 04] KERIEVSKY J., *Refactoring to Patterns*, Addison-Wesley, 1^{st} édition, August 2004.

[KHO 09a] KHOMH F., DI PENTA M., GUÉHÉNEUC Y.-G., "An Exploratory Study of the Impact of Code Smells on Software Change-proneness", ANTONIOL G., ZAIDMAN A., Eds., *Proceedings of the* 16^{th} *Working Conference on Reverse Engineering (WCRE)*, IEEE Computer Society Press, October 2009, 10 pages.

[KHO 09b] KHOMH F., GUÉHÉNEUC Y.-G., ANTONIOL G., "Playing Roles in Design Patterns: An Empirical Descriptive and Analytic Study", KONTOGIANNIS K., XIE T., Eds., *Proceedings of the* 25^{th} *International Conference on Software Maintenance (ICSM)*, IEEE Computer Society Press, September 2009, 10 pages.

[KHO 11] KHOMH F., VAUCHER S., GUÉHÉNEUC Y.-G., SAHRAOUI H., "A GQM-based Method and a Bayesian Approach for the Detection of Code and Design Smells", *Journal of Software and Systems (JSS)*, vol. 84, n°4, Elsevier, April 2011, 35 pages.

[KOR 07] KORU A. G., LIU H., "Identifying and characterizing change-prone classes in two large-scale open-source products", *Journal of Systems and Software*, vol. 80, n°1, Elsevier, January 2007.

[LAN 95] LANGE D. B., NAKAMURA Y., "Interactive Visualization of Design Patterns Can Help in Framework Understanding", *Proceedings of the* 10^{th} *Conference on Object-oriented Programming Systems, Languages, and Applications*, ACM Press, pp. 342–357, 1995.

[MAS 99] MASUDA G., SAKAMOTO N., USHIJIMA K., "Evaluation and Analysis of Applying Design Patterns", ARAKI K., BALZER B., GHEZZI C., KATAYAMA T., KRAMER J., NOTKIN D., PERRY D., Eds., *Proceedings of the* 2^{nd} *International Workshop on the Principles of Software Evolution*, ACM Press, July 1999.

[MCC 77] MCCALL J. A., RICHARDS P. K., WALTERS G. F., "Factors in Software Quality", *National Technical Information Service*, vol. 1, 2, and 3, 1977.

[MCN 01] MCNATT W. B., BIEMAN J. M., "Coupling of Design Patterns: Common Practices and Their Benefits", TSE T., Ed., *Proceedings of the* 25^{th} *Conference on Computer Software and Applications*, IEEE CS Press, pp. 574–579, October 2001.

[MOH 06] MOHA N., GUÉHÉNEUC Y.-G., LEDUC P., "Automatic Generation of Detection Algorithms for Design Defects", UCHITEL S., EASTERBROOK S., Eds., *Proceedings of the* 21^{st} *Conference on Automated Software Engineering*, IEEE Computer Society Press, September 2006, Short paper.

[MOH 08] MOHA N., GUÉHÉNEUC Y.-G., LE MEUR A.-F., DUCHIEN L., "A Domain Analysis to Specify Design Defects and Generate Detection Algorithms", FIADEIRO J., INVERARDI P., Eds., *Proceedings of the* 11^{th} *international conference on Fundamental Approaches to Software Engineering (FASE)*, Springer-Verlag, March-April 2008, 15 pages.

[MOH 10] MOHA N., GUÉHÉNEUC Y.-G., DUCHIEN L., LE MEUR A.-F., "DECOR: A Method for the Specification and Detection of Code and Design Smells", *Transactions on Software Engineering (TSE)*, vol. 36, n°1, IEEE Computer Society Press, January–February 2010, 16 pages.

[NG 07] NG T., CHEUNG S., CHAN W., YU Y., "Do Maintainers Utilize Deployed Design Patterns Effectively?", *Proceedings of the* 29^{th} *International Conference on Software Engineering (ICSE'07)*, IEEE Computer Society, pp. 168-177, 2007.

[OLB 09] OLBRICH S., CRUZES D. S., BASILI V., ZAZWORKA N., "The Evolution and Impact of Code Smells: A Case Study of Two Open Source Systems", *Proceedings of the* 3^{rd} *International Symposium on Empirical Software Engineering and Measurement*, pp. 390–400, October 2009.

[PEA 88] PEARL J., *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufmann, 1^{st} édition, September 1988.

[SHE 11] SHESKIN D. J., *Handbook of Parametric and Nonparametric Statistical Procedures*, Chapman & All, 5^{th} édition, April 2011.

[SHT 01] SHTATLAND E. S., CAIN E., BARTON M. B., "The perils of stepwise logistic regression and how to escape them using information criteria and the Output Delivery System", *Proceedings of the* 26^{th} *SAS Users Group International Conference*, SAS Institute, April 2001.

[VAU 09] VAUCHER S., KHOMH F., MOHA N., GUÉHÉNEUC Y.-G., "Prevention and Cure of Software Defects: Lessons from the Study of God Classes", ANTONIOL G., ZAIDMAN A., Eds., *Proceedings of the* 16^{th} *Working Conference on Reverse Engineering (WCRE)*, IEEE Computer Society Press, October 2009, 10 pages.

[VIN 07] VINAYAGASUNDARAM B., SRIVATSA S., "Software Quality in Artificial Intelligence System", *Information Technology Journal*, vol. 6, n°6, pp. 835-842, 2007.

[VOK 04] VOKAC M., "Defect Frequency and Design Patterns: An Empirical Study of Industrial Code", *Transactions on Software Engineering*, vol. 30, n°12, pp. 904–917, IEEE CS Press, December 2004.

[WEI 07] WEI L., RAED S., "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution", *Journal of Systems and Software*, vol. 80, n°7, Elsevier, 2007.

[WEN 01] WENDORFF P., "Assessment of Design Patterns During Software Reengineering: Lessons Learned from a Large Commercial Project", SOUSA P., EBERT J., Eds., *Proceedings of* 5^{th} *Conference on Software Maintenance and Reengineering*, IEEE Computer Society Press, pp. 77–84, March 2001.

[WYD 98] WYDAEGHE B., VERSCHAEVE K., MICHIELS B., VAN DAMME B., ARCKENS E., JONCKERS V., "Building an OMT-Editor Using Design Patterns: An Experience Report", *Proceedings of the* 26^{th} *Technology of Object-Oriented Languages*, pp. 20–32, August 1998.

[ZHA 04] ZHANG H., "The Optimality of Naive Bayes", BARR V., MARKOV Z., Eds., *Proceedings of the* 17^{th} *International Florida Artificial Intelligence Research Society Conference*, pp. 562–567, May 2004.

[ZIM 07] ZIMMERMANN T., PREMRAJ R., ZELLER A., "Predicting Defects for Eclipse", *Proceedings of the* 3^{rd} *International Workshop on Predictor Models in Software Engineering*, pp. 9–15, May 2007.