Étude de la changeabilité des systèmes orientés objet

Stéphane Vaucher*, Houari Sahraoui*

*Département d'informatique et de recherche opérationnelle,
Université de Montréal
CP 6128 succ. Centre-Ville, Montreal, Quebec, H3C 3J7, Canada
vauchers,sahraouh@iro.umontreal.ca

Résumé. Plusieurs études montrent qu'avec le temps, la plupart des systèmes deviennent difficiles à maintenir et que leur croissance ralentit. Il existe cependant certains systèmes qui utilisent les mécanismes fournis par le paradigme des objets pour soutenir un rythme de développement élevé. Dans cet article, nous étudions les facteurs qui affectent la changeabilité de quatre logiciels libres populaires. Deux applications et deux librairies ont été sélectionnées, puis caractérisées avec des métriques orientées objet classiques. Ces informations ont été utilisées pour bâtir des modèles de prédiction de changement avec des techniques d'apprentissage automatique. Dans le cas de deux librairies avec des modèles de domaine suffisamment précis, les modèles prédictifs ont été capables d'estimer correctement le taux de changement dans le code. Dans le cas de deux applications, ces modèles étaient beaucoup moins précis, mais il a été toutefois possible de prédire les changements dans les classes responsables des interfaces graphiques.

1 Introduction

Nous nous intéressons à l'étude des facteurs qui peuvent influencer l'amplitude des changements dans le code lors de l'évolution des logiciels à objets. En effet, la maintenance des logiciels représente une dépense majeure pour l'industrie (Bell, 2000; Hamlet et Maybee, 2000). Cependant, malgré des avancées importantes en génie logiciel lors des dernières décennies, il est encore difficile d'estimer les coûts de maintenance. Cette situation est d'autant plus préoccupante qu'il existe beaucoup plus de systèmes en maintenance que de nouveaux systèmes développés.

Une façon concrète d'estimer les coûts de maintenance consiste à déterminer a priori, à chaque étape de l'évolution du logiciel (i.e. à chaque version), la quantité de code à changer pour répondre à un ensemble de besoins (Khoshgoftaar et Szabo, 1994; Khoshgoftaar et al., 1996; Nagappan et Ball, 2005). La prédiction des changements dans le code n'est toutefois pas une tâche aisée. En effet, l'état actuel des connaissances ne permet pas de connaître les facteurs qui peuvent influencer positivement ou négativement l'amplitude des changements. Une approche empirique est une bonne alternative, car elle permet d'étudier ces influences en se basant sur des données historiques. Jusqu'à récemment, elle a été utilisée à petite échelle sur des données provenant de systèmes uniques industriels (Li et Henry, 1993; Nagappan et

Ball, 2005). Cependant, la disponibilité récente des logiciels libres permet d'appliquer cette approche à plus grande échelle.

Cet article présente une étude des facteurs influençant la changeabilité d'un logiciel. Selon la norme ISO9126 (ISO9126, 2001), la changeabilité est définie comme la facilité avec laquelle un logiciel peut être modifié pour répondre à un besoin donné, incluant l'ajout de fonctionnalité, la correction d'erreurs et l'adaptation à un nouvel environnement. Cette facilité est en étroite relation avec la quantité de code à changer. En d'autres termes, moins on change de code pour implémenter un besoin, plus le logiciel est réputé facile à modifier.

Selon nous, il existe deux familles de facteurs qui influencent la changeabilité : le besoin de changer et la structure. Plus les besoins sont conséquents, plus on s'attend à un grand changement dans le code. Parallèlement, un système bien structuré devrait pouvoir absorber des nouveaux besoins en minimisant les changements.

L'article est organisé comme suit. Dans la prochaine section, nous présentons ce qu'est un modèle de prédiction ainsi que les mesures que nous utilisons pour bâtir les nôtres. Lors de la section 3, nous présentons une première étude qui évalue la capacité de modèles généraux à prédire les changements. Par la suite, une deuxième étude cherche à montrer si des modèles bâtis sur des systèmes individuels sont plus performants (section 4). Dans une dernière étude (section 5), des modèles sont construits et testés au niveau des sous-systèmes. Les résultats sont finalement discutés et comparés aux travaux connexes (sections 6 et 7).

2 Prédiction des changements de code

Les modèles de prédiction servent à établir une relation entre deux ensembles d'attributs mesurés à différents moments dans le temps. Sur une ligne de temps, deux moments caractérisent un modèle de prédiction : le moment auquel les attributs en entrée sont mesurés et le moment auquel les attributs en sortie peuvent être effectivement mesurés. Ces modèles peuvent alerter une équipe de développement tôt dans le processus de développement de la taille d'un changement, lui permettant d'apporter des corrections si nécessaire.

2.1 Modèle de prédiction

Dans le cadre de ce travail, des modèles sont construits et testés à partir de données historiques. Les modèles prennent en entrée deux types d'information, la structure du système (voir section 2.2) et les changements des besoins (voir section 2.3). Ils produisent en sortie la quantité de code changé (voir section 2.4).

Différents aspects de la structure d'un logiciel comme le couplage et la cohésion, peuvent être mesurés directement à partir du code, une source d'information fiable. Ce n'est toutefois pas le cas pour les changements dans les besoins. En effet, les documents décrivant les besoins sont souvent manquants ou incomplets. Nous utilisons donc la prochaine source d'information disponible, soit la conception. Les changements dans la conception peuvent être mesurés en comparant la conception présente à une nouvelle conception proposée. Le modèle dans la figure 1 montre les moments auxquels les artefacts des phases d'implémentation ou de conception sont produits par rapport aux entrées et aux sorties du modèle de prédiction.

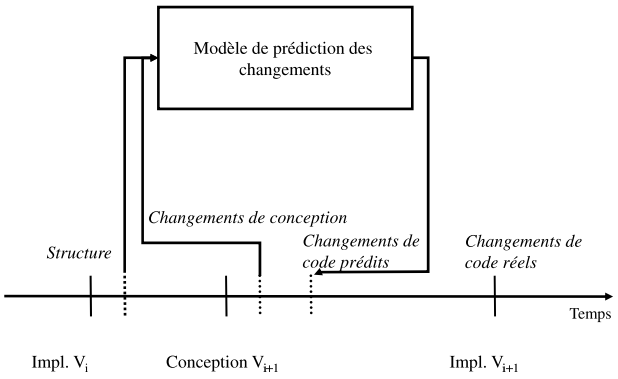


FIG. 1 – *Modèle de prédiction du changement*

2.2 Métriques de structure (entrée 1)

Pour mesurer les caractéristiques structurelles des systèmes, nous avons utilisé des métriques statiques pour quantifier la taille, la complexité, le couplage, la cohésion et l'utilisation d'héritage. Ces métriques ont été bien étudiées, entre autres, dans (Li et Henry, 1993; Basili et al., 1996; Briand et al., 1997).

Métriques de couplage - Nous avons utilisé trois métriques pour mesurer différents types de couplage. Pour chaque classe, CBO compte le nombre de classes qui sont utilisées par elle ou qui l'utilisent. Les liens d'utilisation représentent soit un accès aux attributs, soit un envoi de message. Nous avons également deux mesures pour indiquer le couplage dans l'arbre d'héritage. Des mesures comptent le nombre de sous-classes qui invoquent les méthodes d'une classe (DCMED) et qui accèdent à ses attributs (DCAEC) (Briand et al., 1997).

Métriques de cohésion - Deux catégories de cohésion (ou manque de cohésion) sont mesurées. LCOM5 (Chidamber et Kemerer, 1991; Henderson-Sellers, 1996) trouve le ratio d'attributs qui ne sont pas utilisés par les méthodes d'une classe ; ICH (Y. S. Lee, 1995) calcule le degré d'utilisation des méthodes d'une classe.

Métriques d'héritage - DIT (Chidamber et Kemerer, 1991) mesure la distance entre une classe et la racine dans son arbre d'héritage. La métrique ABS que nous avons définie correspond à la proportion du nombre de méthodes abstraites par rapport au nombre total de méthodes fournies par une classe.

Métriques de taille et de complexité - WMC (Chidamber et Kemerer, 1991) définit la complexité d'une classe comme la somme de la complexité de ses méthodes. Dans cette étude, nous avons utilisé la complexité cyclomatique de McCabe (McCabe, 1976) des méthodes. NMD, NAD et NI comptent respectivement le nombre de méthodes, d'attributs et d'instructions déclarés dans une classe.

2.3 Changements de conception (entrée 1)

Dans le paradigme objet, l'interface d'une classe définit les services qu'elle offre ; elle est donc une partie importante de la conception d'un système. Nous avons donc décidé de quantifier les changements dans l'interface avec deux métriques adaptées de celles décrites dans. Grosser et al (Grosser et al., 2003) : la Proportion de Méthodes Publiques Ajoutées (PMPA, équation 1) et la Proportion des Méthodes Publiques Retirées (PMPR, équation 2). Dans les deux équations, $I(N)$ est l'ensemble de méthodes publiques déclarées et héritées d'une classe lors de la version N. Ces mesures sont normalisées par la taille maximale de l'interface pour les borner dans l'intervalle [0,1] . Les changements de signature ne sont pas pris en compte. Ils sont considérés à la fois comme un ajout et une suppression.

$$PMPA = \frac{|I(N+1) - I(N)|}{MAX(|I(N)|, |I(N+1)|)} \quad (1)$$

$$PMPR = \frac{|I(N) - I(N+1)|}{MAX(|I(N)|, |I(N+1)|)} \quad (2)$$

class C public m1() ; public m2() ; public m3() ;	class C public m3() ; public m4() ;

TAB. 1 – *Deux versions consécutives de la classe C*

L'exemple de la table 1 montre une classe contenant trois méthodes dans la version i. Seule une (m3) demeure dans la version $i+1$. Sur cet exemple, nous retrouvons : $PMPA = 1/3$ et $PMPR = 2/3$.

2.4 Changements de code (sortie)

Le changement dans les lignes de code a souvent été utilisé pour mesurer un effort de maintenance (Khoshgoftaar et Szabo, 1994; Khoshgoftaar et al., 1996; Nagappan et Ball, 2005). Dans la plupart des cas, les chercheurs utilisent l'utilitaire Unix *diff* pour mesure les changements. Un problème avec cette approche est qu'elle surestime l'amplitude de certains changements comme un reformatage.

Nous mesurons le changement de code à partir des changements en instructions. Pour toute méthode présente dans deux versions consécutives d'un logiciel et partageant une même signature, nous calculons une distance de Levenshtein (Levenshtein, 1966). La distance compte le nombre d'instructions ajoutées, supprimées et modifiées. Si une méthode a été supprimée, elle est traitée comme une méthode qui ne contient aucune instruction. Le changement total d'une classe est la somme des changements dans chaque méthode normalisé par la taille maximale de la classe. Par exemple, si la séquence d'instructions :

A	B	C	D	A

devient

A	B	B'	D	D'	F

où les lettres sont des instructions. L'algorithme de Levenshtein trouve l'alignement suivant avec un ratio de changement de 50% :

A	B	C	D		A
A	B	B'	D	D'	F
=	=	Δ	=	+	Δ

Dans le cas de cette étude, les outils sont adaptés au langage Java et nous utilisons les changements des instructions en code octet. Pour éviter qu'il y ait des erreurs provenant de la mesure des changements, le même compilateur a été utilisé pour l'analyse de tous les systèmes sans aucune optimisation. Dans cette situation, nous avons trouvé que les tailles du code source et du code octet sont corrélées à 0.98.

3 Étude générale de la changeabilité

Cette section présente l'étude de la changeabilité des classes utilisant la totalité des données que nous avons récoltées. Elle consiste à vérifier si un modèle obtenu à partir de ces données offre une bonne capacité de prédiction.

3.1 Conception de l'étude

Dans cette étude, trois modèles sont construits et testés sur des données provenant de quatre différents systèmes. Ces modèles vérifient l'influence des changements dans la conception et l'influence de la structure sur les changements dans le code. Le premier modèle (M1) utilise seulement les changements de conception en entrée et quantifie leur influence sur les changements dans le code. Le deuxième modèle (M2) cherche à établir une relation entre la structure des classes et les changements. Le dernier modèle (M3) inclut les deux types d'information pour voir si leur combinaison améliore la prédiction des changements. Ce dernier modèle est conçu pour valider l'intuition selon laquelle un système bien structuré devrait pouvoir absorber facilement les changements de conception et donc évoluer à peu de coût.

3.1.1 Systèmes étudiés

Les systèmes étudiés sont les suivants : Azureus, ArgoUML, Xerces-J et Xalan-J. Tous les systèmes ont des communautés d'utilisateurs et de développeurs actifs depuis plusieurs années. Deux des systèmes sélectionnés, Azureus et ArgoUML, sont des applications avec interfaces graphiques tandis que Xerces-J et Xalan-J sont des librairies.

Les versions mineures de chaque système ont été extraites des gestionnaires de versions. Au total, l'étude porte sur 24 000 évolutions où une évolution est une transition d'une classe d'une version à une autre. Chaque système à l'exception d'Azureus contribue pour environ 4 000 évolutions. Azureus est trois fois plus grand que les autres. Ainsi, à cause de sa grande taille, ses sous-systèmes feront l'objet d'une étude détaillée.

Des métriques descriptives de la structure des classes sont présentées dans la table 2. Il est possible de voir qu'en général, la plupart des classes implémentent une dizaine de méthodes (NMD) et utilisent des relations d'héritage (médiane de DIT > 1). La majorité des classes (> 80%) présente peu ou pas de changements dans la conception et dans le code. Ceci est

Métrique	Min	Médiane	Max
CBO	1	7	12
DCMEC	0	0	6
DCAEC	0	0	8
LCOM5	0	0,67	2
ICH	0	2	2198
DIT	1	2	10
ABS	0	0	1
WMC	0	10	801
NMD	1	8	147
NAD	0	5	37
NI	1	98	17404

TAB. 2 – *Valeurs des métriques de structure (min, médiane et max)*

cohérent avec l'opinion de Boehm (Boehm et Papaccio, 1988) qui mentionne que peu de composants sont responsables de la majorité des coûts de maintenance. Un modèle prédisant des changements de cette distribution doit donc être capable de distinguer entre les classes qui changent beaucoup et celles qui changent peu.

3.1.2 Technique d'analyse

Les modèles ont été construits avec des arbres de régression (Breiman et al., 1984). Les arbres de régressions sont une technique d'apprentissage de type diviser pour régner qui dérive des règles de prédictions en les organisant sous forme d'arbre. Chaque nœud contient une règle tandis que chaque feuille garde une valeur utilisée pour la prédiction. Un avantage de cette technique est qu'elle génère une abstraction de la relation. Comme les décisions sont explicites, elles peuvent être analysées quand le modèle démontre une bonne capacité prédictive.

Dans les modèles de régression, il est normal d'utiliser l'erreur quadratique moyenne pour juger de leur qualité, mais cette mesure surestime la capacité d'un modèle quand la majorité des valeurs de changement sont petites. La corrélation est une bonne alternative pour juger la performance des modèles. Quand la corrélation est élevée, le modèle est capable de prédire l'amplitude des changements. Les jeux de données que nous utilisons présentent un haut degré de liberté, car ils contiennent minimalement 3000 cas. En conséquence et comme il est d'usage, nous considérons une corrélation d'au moins 0,5 comme statistiquement significative.

Pour maximiser l'utilisation des données, les modèles sont bâtis et testés avec une validation croisée à 10 plis. Cette approche est assez standard en apprentissage et consiste à diviser les données en 10 plis (ou groupes). Un modèle est bâti sur chaque combinaison de 9 plis et il est testé sur le pli restant. Le résultat total de la prédiction est la moyenne des 10 modèles.

3.2 Résultats et interprétation

Les modèles généraux sont bâtis et testés sur les 24 000 évolutions provenant de tous les systèmes. La table 3 présente les corrélations pour les modèles obtenus. Les métriques de changements de conception (M1) et les métriques de structure (M2) fournissent assez d'éléments

pour permettre une bonne prédiction des changements dans le code, affichant des corrélations autour de 0,6. Avec une corrélation de 0,79, la combinaison des deux types de métriques permet une prédiction nettement meilleure.

Algorithme	M1	M2	M3
Arbres de régression	0,63	0,59	0,79

TAB. 3 – *Résultats des modèles généraux (corrélation)*

Les résultats sont illustrés dans la figure 2 qui est composée de paires de points alignés verticalement correspondant aux valeurs prédites (point gris) et aux valeurs réelles (point noir) de chaque évolution. Ces points sont affichés en ordre des valeurs réelles. L'erreur de prédiction est la distance verticale séparant les points gris des points noirs. La figure 2(a) montre qu'à partir des changements de conception, il est possible d'identifier quand un changement se produira sans nécessairement prédire correctement sa magnitude. Pour les métriques de structure (figure 2(b)), c'est plutôt l'opposé qui se produit. Le modèle est capable d'estimer l'ampleur des changements dans les classes fortement changées, mais il est incapable de trouver quand un petit changement se produira. Finalement, le dernier modèle (M3) semble correctement prédire à la fois les petits changements et les grands changements.

Ces résultats sont prometteurs, mais l'analyse détaillée des classes ayant présenté des erreurs de prédiction nous indique que la nature des systèmes devrait être prise en compte. Ainsi, nous avons décidé d'étudier la changeabilité de chaque système séparément.

4 Étude de l'évolution par système

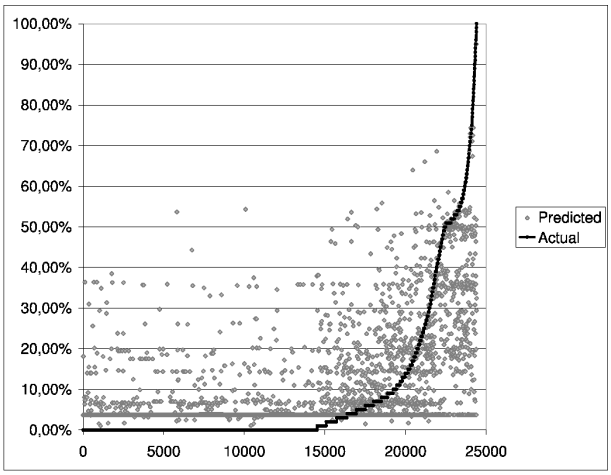
Les modèles généraux n'incluent aucune information sur le type de système traité. Ainsi, dans une deuxième étude qui suit la même méthodologie que l'étude précédente, les systèmes ont été analysés séparément. Dans un premier temps, nous présentons les différentes données, puis nous discutons des résultats obtenus.

4.1 Systèmes étudiés

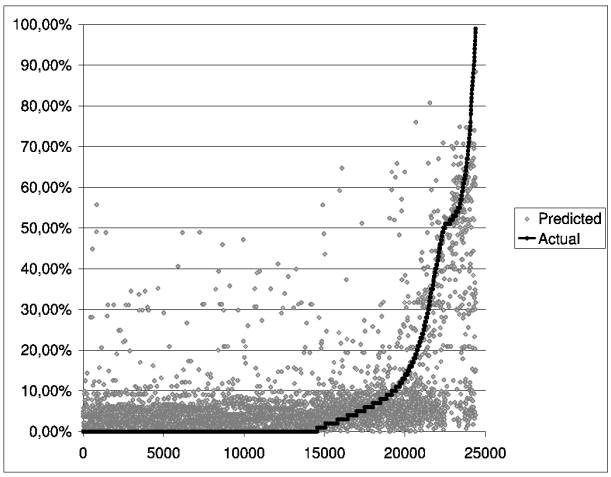
Chaque système présente des caractéristiques structurelles qui peuvent être utilisées pour bâtir des modèles spécialisés. La table 4 donne les valeurs moyennes des métriques de structure des classes de chaque système. Nous remarquons que les deux librairies sont constituées de classes plus grosses que les classes des applications à la fois en complexité (WMC) et en nombre d'instruction (NI). De plus, Azureus est le seul système qui utilise peu l'héritage avec des classes qui se situent à une profondeur moyenne de 1,4 dans l'arbre d'héritage.

4.2 Résultats et interprétation

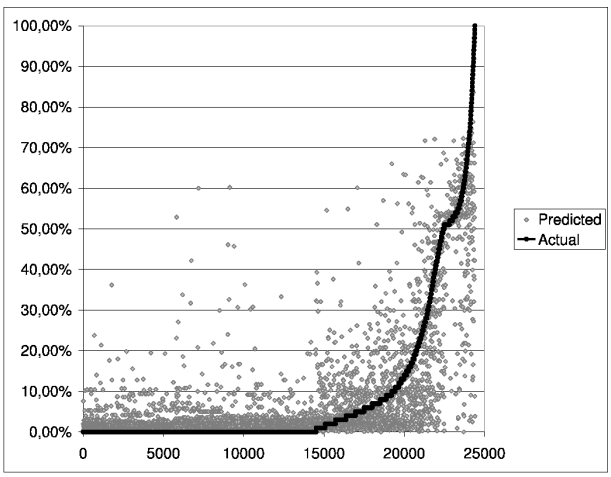
Le tableau 5 présente les résultats de chaque modèle. Nous interpréterons les résultats en fonction des trois types de modèle.



(a) M1



(b) M2



(c) M3

FIG. 2 – *Taux de changement de code prédit versus réel*

Métrique	Xerces	Xalan	Azureus	ArgoUML
CBO	18	20,5	17,5	158,2
DCMEC	0,03	0,02	0	0,01
DCAEC	0,02	0,04	0	0,01
LCOM5	0,62	0,5	0,53	0,45
ICH	20,1	9,35	11,4	12,6
DIT	3	2,7	1,63	2,8
ABS	0,21	0,01	0	0,01
WMC	38,6	27,1	18,9	19,9
NMD	12,7	9,35	8,45	8,2
NAD	6,7	5,7	4,9	3,7
NI	413	372	265	212

TAB. 4 – *Valeur moyenne des métriques de structure par système*

Système	M1	M2	M3
Xerces	0,63	0,71	0,82
Xalan	0,48	0,63	0,76
ArgoUML	0,53	0,21	0,56
Azureus	0,71	0,21	0,72
Moyenne	0,59	0,39	0,70

TAB. 5 – *Résultats par système (corrélation)*

Modèles 1 : Prédiction du changement dans le code à partir des changements de la conception. Les résultats montrent qu'en général, les changements de conception sont utiles (corrélation de 0.59 en moyenne) pour estimer les changements dans le code. Ce qui est surprenant, est toutefois la faible corrélation chez Xalan. Ce système contient plusieurs grandes classes instables qui implémentent peu de méthodes et leurs interfaces ne changent jamais. Le modèle pour Azureus produit le meilleur résultat parce que l'application utilise peu l'héritage, et que tout changement à ses interfaces entraîne un changement presque équivalent dans le code.

Modèles 2 : La structure comme prédicteur des changements dans le code. Les résultats montrent que la prédiction est bonne pour les librairies, mais pas pour les applications. De manière générale, il y a deux raisons qui expliquent pourquoi la structure affecterait l'évolution d'un système. Premièrement, une bonne structure utilisant les mécanismes orientés objet peut limiter l'ampleur des changements. Deuxièmement, il se peut qu'une mauvaise structure puisse augmenter les coûts d'apporter un changement.

Dans les systèmes orientés objet, l'héritage permet de réutiliser du code dans différentes classes. Les librairies contiennent des arbres d'héritage profonds pour modéliser leurs domaines. Les classes offrant des fonctionnalités réutilisables sont généralement proches de la racine et elles changent rarement. Les classes profondes quant à elles offrent des fonctionnalités susceptibles de changer. Par exemple, 25% des classes de Xerces sont à une profondeur

d'au moins 5 de la racine de l'arbre d'héritage et la majorité des changements prédits correctement y sont situés. ArgoUML utilise aussi l'héritage, mais n'inclut pas un modèle de domaine bien structuré qui semble être le facteur permettant de prédire le changement.

La cohésion des classes de Xalan semble être un autre facteur important. Quand une classe contient des méthodes utilisant les mêmes attributs, elle devrait être plus facile à maintenir. Un groupe de classes peu cohésives identifiées à cause d'un grand manque de cohésion (LCOM5 de 1,5 versus 0,5) contient les classes les plus changées du système.

Finalement, nous avons remarqué que le couplage et la taille influencent peu la changeabilité des librairies. Les seuls cas où ces métriques participent à la prédiction, sont des grandes classes peu couplées et qui contiennent de petits changements.

Modèles 3 : La combinaison des changements de conception et la structure pour la prédiction du changement dans le code. Les modèles des librairies démontrent une amélioration marquée par rapport aux deux modèles obtenus avec seulement une des deux entrées possibles. Ainsi, nous pouvons affirmer que l'utilisation des deux types de métriques améliore la précision d'un modèle portant sur les librairies, mais pas sur les applications. Ceci n'est pas surprenant si l'on considère la faible influence de la structure telle que montrée dans les modèles de type 2.

Une lacune avec les modèles construits par système est que la structure des applications ne fournit aucune information pour permettre de comprendre son évolution. Étant donné qu'Azureus comprend plus de 13 000 évolutions, dans une dernière étude, nous allons voir si le rôle qu'a une classe dans le système peut améliorer la prédiction en étudiant l'évolution par sous-système.

5 Étude des sous-systèmes

La dernière étude cherche à vérifier l'évolution de différents sous-systèmes d'Azureus. À chaque nouvelle version, les développeurs d'Azureus présentent les changements par rapport au sous-système affecté. Ils indiquent systématiquement si un changement affecte le noyau ou les interfaces. Nous avons donc gardé cette organisation pour cette étude.

La table 6 présente les résultats des modèles obtenus sur les différents sous-systèmes. Les modèles de type 2 (M2) montrent que les changements dans les interfaces graphiques d'un système sont affectés par sa structure (corrélation de 0,62), mais ceci n'est pas le cas pour le noyau (corrélation de 0,11). Ceci est explicable par l'utilisation de relations d'héritage comme présenté dans la table 7. En fait, seulement 20% des classes du noyau utilisent l'héritage contre plus de 50% des classes d'interface.

Configuration (nombre de cas)	M1	M2	M3
Total (13 084)	0,71	0,21	0,72
Noyau (4 310)	0,68	0,11	0,68
Interfaces (1 217)	0,83	0,62	0,84

TAB. 6 – *Corrélation des sous-systèmes d'Azureus*

Système	DIT moyen
Azureus (total)	1,63
Azureus (interfaces)	2,27
Azureus (noyau)	1,37

TAB. 7 – *Héritage dans Azureus*

Les interfaces graphiques utilisent une architecture MVC pour fournir plusieurs vues de l'application (HTML, applets, SWT). Or, cette architecture est bien connue par la communauté orientée objet. Il n'est donc pas surprenant que cette partie du système utilise l'héritage pour réutiliser du code et minimiser les changements. Dans ce cas, M2 utilise la structure pour identifier quel est le rôle d'une classe dans le système et cette information améliore sa changeabilité (corrélation de 0,62).

6 Travaux connexes

La majorité de la littérature sur la modélisation de la changeabilité porte soit sur les modèles d'impact de changements (Arnold, 1996; Chaumun et al., 1999), soit sur son utilisation comme mesure de l'effort de maintenance. Ce travail se situe dans la deuxième catégorie. Plusieurs travaux vérifient l'impact de changer un module sur sa propension à engendrer des erreurs (Khoshgoftaar et al., 1996; Munson et Elbaum, 1998; Nagappan et Ball, 2005). En particulier, il a été observé que les modules qui changent beaucoup ont plus de chance de contenir des erreurs.

La première étude identifiant des facteurs influençant la changeabilité de systèmes OO vient de Li et al. (Li et Henry, 1993). Elle a montré que la structure d'un système affecte ses coûts de maintenance, mesurés comme des changements en lignes de code. Ce travail n'a toutefois pas expliqué les raisons de cette relation. Nous avons proposées l'idée que la structure permet d'identifier le rôle de certaines classes et que cette information permet de prédire l'amplitude des changements dans un système. Dans (Li et al., 2000), les auteurs ont présenté des métriques de changements orientées objet. Étudiant le développement incrémental d'un système de petite taille (20 classes), les auteurs ont montré que métriques de changement (conception et implantation) et métriques de structure sont indépendantes. En analysant plusieurs systèmes de taille moyenne, nous avons trouvé le contraire : il existe une relation d'influence quand les développeurs ont utilisé des mécanismes orientés objet. Dans (Mao et al., 1998), les auteurs ont présenté une étude dans laquelle ils ont évalué l'influence de l'héritage sur la réutilisabilité d'un logiciel. Ils ont demandé à un développeur d'estimer les coûts d'adaptation d'une librairie à un nouveau domaine. Ils ont montré que l'héritage est un facteur important pour encourager la réutilisation et minimiser les changements. Ceci est cohérent avec nos résultats.

Récemment, plusieurs travaux (Bouktif et al., 2006; Gîrba et al., 2007) ont porté sur la découverte de groupes de classes qui changent en même temps (co-changements). Dans notre cas, les algorithmes d'apprentissages ont été capables de découvrir automatiquement certaines structures dans le code identifiant certains de ces groupes et d'inclure cette information dans les modèles prédictifs.

7 Discussion et conclusions

Dans cet article, nous avons présenté trois études ayant comme but d'identifier les facteurs affectant la changeabilité des logiciels orientés objet. Nous avons utilisé une technique d'apprentissage pour construire des modèles de prédiction qui ont été évalués sur quatre systèmes industriels. Les études utilisaient trois types de modèles pour vérifier si le changement dans le code pouvait être prédit par les changements de conception (M1), la structure (M2) et si une combinaison des deux facteurs pouvait améliorer la prédiction (M3). De manière générale, les modèles produisent des prédictions qui sont fortement corrélées aux changements réels observés, mais le succès dépend de l'usage correct de mécanismes orientés objet. Les résultats peuvent être récapitulés comme suit :

- M1 : Les changements dans le code dépendent fortement des changements de la conception. Ces modèles fonctionnent bien quelle que soit la nature du système, mais leur capacité de prédiction est limitée par la simplicité des mesures utilisées. Ceci est évident lorsqu'on observe une classe dont l'interface n'a pas changé ;
- M2 : Les métriques de structure contribuent à la prédiction des changements à condition que le système suive les principes du paradigme objet. Ceci a été observé à la fois dans Xerces et Xalan, ainsi que dans les interfaces graphiques d'Azureus. Les métriques les plus importantes étaient DIT (profondeur dans l'arbre d'héritage) et LCOM5 (manque de cohésion) ;
- M3 : Quand les modèles de structure fonctionnent, alors il est possible de combiner les deux types de métriques pour améliorer la précision des prédictions ;
- Dans certains cas, les modèles généraux donnaient de meilleurs résultats que les modèles des systèmes individuels. Nous croyons qu'il pourrait être possible d'utiliser un modèle général pour traiter un nouveau système en l'absence de données historiques ;
- Quand un modèle est incapable de prédire les changements dans un système, les changements dans certains sous-systèmes peuvent être prédits comme observé avec les interfaces d'Azureus.

Plusieurs facteurs peuvent menacer la validité des études et limiter la généralisation des résultats. Premièrement, il y a le choix des systèmes. Les systèmes ont été sélectionnés en fonction de leur nature (librairie vs application), leur popularité et leur maturité. Le domaine cependant n'a pas été pris en compte, car trois des quatre systèmes sont des outils de développement. Cependant, les 24 000 évolutions analysées devraient contenir suffisamment de variations en structure et changements.

Il y a également la mesure de performance des modèles. La corrélation n'indique pas précisément si un modèle produit des erreurs, mais plutôt la capacité d'un modèle à prédire la magnitude des changements. Les résultats présentés ont également été corroborés par des mesures d'erreur quadratique.

Nous avons identifié deux axes de recherche pour nos travaux futurs. Premièrement, nous sommes intéressés à comprendre comment l'architecture d'un logiciel affecte sa changeabilité. Entre autres, nous croyons qu'elle peut nous aider à identifier la nature de certaines classes, un facteur important pour prédire les changements. Deuxièmement, nos résultats nous laissent croire qu'il existerait des patrons d'évolution autres que des co-changements. Nous planifions appliquer des techniques de fouille de données pour explorer cette idée.

8 Remerciements

Cette recherche a été financée partiellement par le Fond québécois de recherche sur la nature et les technologies (FQRNT) et le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG).

Références

Arnold, R. S. (1996). *Software Change Impact Analysis*. Los Alamitos, CA, USA : IEEE Computer Society Press.

Basili, V. R., L. C. Briand, et W. L. Melo (1996). A validation of object-oriented design metrics as quality indicators. *Software Engineering 22*(10), 751–761.

Bell, D. (2000). *Software Engineering : A Programming Approach*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc.

Boehm, B. W. et P. N. Papaccio (1988). Understanding and controlling software costs. *IEEE Transactions Softw. Eng. 14*(10), 1462–1477.

Bouktif, S., Y.-G. Guéhéneuc, et G. Antoniol (2006). Extracting change-patterns from cvs repositories. In *WCRE '06 : Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, Washington, DC, USA, pp. 221–230. IEEE Computer Society.

Breiman, L., J. H. Friedman, R. A. Olshen, et C. J. Stone (1984). *Classification and Regression Trees*. Statistics/Probability Series. Belmont, California, U.S.A. : Wadsworth Publishing Company.

Briand, L. C., P. T. Devanbu, et W. L. Melo (1997). An investigation into coupling measures for C++. In *International Conference on Software Engineering*, pp. 412–421.

Chaumun, M. A., H. Kabaili, R. K. Keller, et F. Lustman (1999). A change impact model for changeability assessment in object-oriented software systems. In *CSMR '99 : Proceedings of the Third European Conference on Software Maintenance and Reengineering*, Washington, DC, USA, pp. 130. IEEE Computer Society.

Chidamber, S. R. et C. F. Kemerer (1991). Towards a metrics suite for object oriented design. In *OOPSLA '91 : Conference proceedings on Object-oriented programming systems, languages, and applications*, New York, NY, USA, pp. 197–211. ACM Press.

Gîrba, T., S. Ducasse, A. Kuhn, R. Marinescu, et R. Daniel (2007). Using concept analysis to detect co-change patterns. In *IWPSE '07 : Ninth international workshop on Principles of software evolution*, New York, NY, USA, pp. 83–89. ACM.

Grosser, D., H. A. Sahraoui, et P. Valtchev (2003). An analogy-based approach for predicting design stability of java classes. In *Proceedings of the 9th International Symposium on Software Metrics*, Washington, DC, USA, pp. 252. IEEE Computer Society.

Hamlet, D. et J. Maybee (2000). *The Engineering of Software : A Technical Guide for the Individual*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc.

Henderson-Sellers, B. (1996). *Object-oriented metrics : measures of complexity*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc.

ISO9126 (2001). ISO/IEC 9126-1 :2001, software engineering – product quality – part 1 : Quality model.

Khoshgoftaar, T., E. Allen, N. Goel, A. Nandi, et J. McMullan (1996). Detection of software modules with high debug code churn in a very large legacy system. *ISSRE '96 : Proceedings of the Eighth International Symposium on Software Reliability Engineering 00*, 364.

Khoshgoftaar, T. M. et R. M. Szabo (1994). Improving code churn predictions during the system test and maintenance phases. In *Proceedings of the International Conference on Software Maintenance*, Washington, DC, USA, pp. 58–67. IEEE Computer Society.

Levenshtein, V. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Journal of Soviet Physics - Doklady 10*(8), 707–710.

Li, W., L. H. Etzkorn, C. G. Davis, et J. R. Talburt (2000). An empirical study of object-oriented system evolution. *Information and Software Technology 42*, 373–381.

Li, W. et S. Henry (1993). Object-oriented metrics that predict maintainability. *Journal of Software Systems 23*(2), 111–122.

Mao, Y., H. Sahraoui, et H. Lounis (1998). Reusability hypothesis verification using machine learning techniques : A case study. In *Proceedings of the 13th IEEE international conference on Automated software engineering*, Washington, DC, USA, pp. 84. IEEE Computer Society.

McCabe (1976). A complexity measure. *IEEE Transactions Softw. Eng. 2*, 308–320.

Munson, J. C. et S. G. Elbaum (1998). Code churn : A measure for estimating the impact of code change. In *ICSM '98 : Proceedings of the international conference on software maintenance*, Washington, DC, USA, pp. 24. IEEE Computer Society.

Nagappan, N. et T. Ball (2005). Use of relative code churn measures to predict system defect density. In *ICSE '05 : Proceedings of the 27th international conference on Software engineering*, pp. 284–292.

Y. S. Lee, B. S. L. (1995). Measuring the coupling and cohesion of an object-oriented program based on information flow. Maribor, Slovenia, pp. 81–90.

Summary

Studies show that with time, most software systems become harder to maintain and their growth slows down. There exists however certain systems that use mechanisms offered by object-oriented languages to sustain a healthy growth rate. In this paper, we study the factors that affect the changeability of four open-source software systems. Two applications and two librairies were selected and characterised using classic OO metrics. Using machine learning techniques, this information was used to build models capable of predicting code change. With the two librairies with well-structured domain models, the predictive models were able to correctly assess relative code changes. On the other hand, the models for the applications were not able to predict code change well in all classes, only in the classes responsible for user interfaces.