TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

API	Application Programming Interface
OO	Objected Oriented
ISP	Interface Segregation Principle
GoF	Gang Of Four
OCP	Open Closed Principle
SVN	SubVersioN
CSV	Comma Separated Values
CVS	Concurrent Versions System
NoC	Number Of Children
UML	Unified Modeling Language

INTRODUCTION

Contexte et problématique

L'évolution des logiciels est reconnue comme un processus naturel pour les systèmes d'une longue durée de vie (Lehman, 1980). Les raisons de l'évolution sont principalement liées aux changements dus à l'ajout ou l'amélioration des fonctionnalités, la correction des bogues, la refactorisation du code source, l'adaptation à un nouvel environnement, etc. Bien que ces changements soient nécessaires pour garder la pertinence des logiciels, l'évolution peut mener à la dégradation de la conception, et rend le système plus difficile à maintenir et à tester. En conséquence, les logiciels peuvent devenir ingérables et éventuellement inutilisables (Van Gurp et Bosch, 2002).

En génie logiciel, les abstractions sont des mécanismes importants dans la conception permettant de gérer les parties complexes des systèmes et de faciliter le développement modulaire. L'idée fondamentale de ce concept revient au principe de masquage des informations proposé par Parnas (1972) qui consiste en la création des modules permettant de dissimuler les décisions de conception qui sont difficiles ou présentent des possibilités de changement. Une abstraction peut prendre la forme d'une interface entre les détails des implémentations d'un module souvent exposés aux changements et les autres composants d'un système. Le concepteur décide les fonctionnalités qui sont visibles à l'extérieur et qui forment les abstractions des variations. L'avantage de ce mécanisme est de faciliter le développement parallèle en équipe. Les composants logiciels peuvent évoluer de façon indépendante. De plus, il favorise le faible couplage entre les modules en minimisant les dépendances risquées.

Les langages modernes de la programmation orientée objet prennent en charge l'abstraction à travers plusieurs techniques de développement. En Java, les interfaces sont des concepts importants permettant de cacher les implémentations par rapport aux composants externes appelés clients. Une interface déclare un ensemble de contrats de service qui définissent la façon d'interaction entre les modules logiciels. La protection de variations (Larman, 2001) ou de détails ne peut être achevée que si les clients dépendent seulement de l'interface. La minimisation des

couplages avec les artefacts instables cachés derrière l'interface favorise l'isolation des parties complexes du système, la facilité de l'extension des fonctionnalités et la stabilité des clients en diminuant l'impact des changements inévitables des implémentations.

Les chercheurs et les experts ont proposé une variété de techniques de développement promouvant une meilleure utilisation de ce mécanisme. La programmation à l'interface et non aux implémentations proposée par Gamma *et al.* (1994) constitue un concept fondamental dans l'utilisation des interfaces. L'idée derrière cela est que les clients dépendent des interfaces et n'invoquent pas les variations isolées. Ce type de couplage est considéré comme stable parce que les clients connaissent uniquement les abstractions. Les clients sont privés de la création des objets dérivés de l'interface. Cette responsabilité peut être affectée à un patron créationnel tel que les fabriques dans GoF (Gamma *et al.*, 1994).

Cependant, les défauts de conception dans les interfaces peuvent mener à une augmentation des coûts de développement, et à la dégradation de la qualité du code (Abdeen *et al.*, 2013a). Le couplage des clients avec les implémentations est jugé coûteux (Izurieta et Bieman, 2013) et implique que ces derniers ne sont pas protégés aux changements qui se produisent au niveau des variations. Ce type de dépendance est instable, car il peut introduire des changements dans le système et augmente le nombre du couplage risqué. Dans la pratique, les développeurs des classes clients invoquent directement les implémentations des interfaces sans considérer les risques liés à l'ajout de telles dépendances sur la stabilité des conceptions. Puisque le contrôle d'accès strict au niveau de classe logicielle n'est pas couramment utilisé par les développeurs, le principe de masquage des détails est souvent non respecté (Rufiange et Fuhrman, 2014). En conséquence les clients ayant des couplages vers les implémentations peuvent être exposés aux changements et aux défauts de conception.

Dans la littérature, plusieurs travaux ont été effectués sur les interfaces en programmation orientée objet en ce qui concerne les anomalies dans leur conception (Abdeen *et al.*, 2013a), les métriques des mesures de la complexité des interfaces et leur prédisposition aux changements (Romano et Pinzger 2011 ; Abdeen *et al.* 2013b ; Cataldo *et al.* 2010 ; Boxall et Araban

2004). Toutefois, très peu d'études ont été réalisées sur l'impact de l'existence des couplages entre les clients et les détails des interfaces en ce qui concerne le risque des changements dans l'évolution des logiciels. Dans ce contexte, il est nécessaire de réaliser des études empiriques visant à analyser les conséquences de l'introduction des dépendances avec les implémentations sur la prédisposition des clients des interfaces aux changements et aux bogues.

Objectifs

Dans cette étude, nous cherchons à analyser l'impact de l'existence des couplages risqués sur la stabilité des clients des interfaces. Nous proposons donc une validation empirique visant à déterminer dans quelle mesure les dépendances avec les implémentations peuvent exposer les clients aux changements. En utilisant des techniques d'extraction de données à partir des systèmes de contrôle de versions et d'identification des clients des interfaces ayant des couplages stables (les classes clients censées être protégées des changements et qui dépendent seulement des interfaces) et des couplages instables (les classes clients qui dépendent simultanément des interfaces et leurs implémentations), nous cherchons à évaluer empiriquement s'il y a une différence entre les clients stables et les clients instables en ce qui concerne leur prédisposition aux changements et aux bogues dans l'évolution des systèmes étudiés.

Dans ce projet, nous limitons notre approche à l'étude du mécanisme d'interface comme une technique de protection des variations. De plus, nous limitons l'analyse au niveau de la collecte de données à des projets de code source ouvert, écrits en Java et hébergés dans les logiciels de gestion des versions Subversion [1] ou Git [2].

Les contributions générales de cette étude sont l'exploration et la validation empirique de l'évolution des structures de conception supportant la variabilité. Nous présentons plus d'aperçus sur les conséquences de violation du principe de masquage d'information des interfaces sur l'introduction des changements.

1. http://subversion.apache.org
2. http://git-scm.com

Méthodologie de recherche

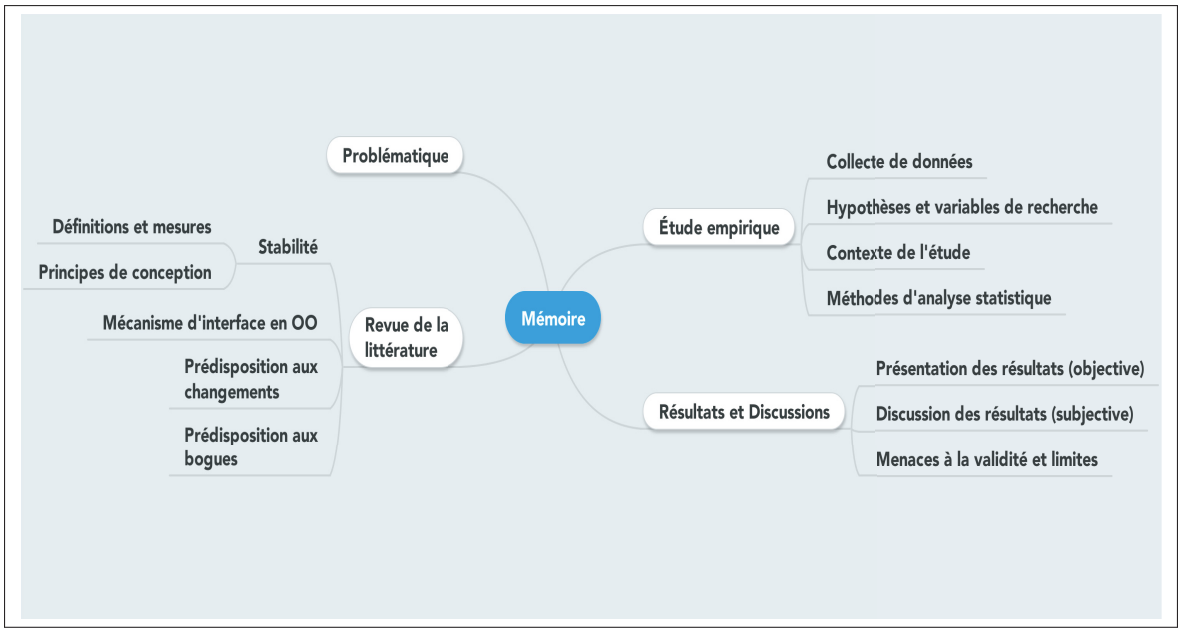

Figure 0.1 Structure du mémoire sous forme de carte conceptuelle.

Pour atteindre nos objectifs, nous avons suivi une méthodologie basée sur la structure présentée dans la Figure 0.1. En premier lieu et après la définition de notre problématique, nous avons réalisé une phase de lecture et d'analyse des principaux travaux antérieurs liés à notre sujet. Nous avons étudié la stabilité des logiciels dans la littérature et les principes de conception proposés pour promouvoir cette qualité et la capacité à gérer les changements au fil du temps. Nous avons aussi effectué un survol sur le mécanisme d'interface dans la conception orientée objet et les différentes techniques favorisant l'utilisation efficace de ce mécanisme, ainsi que sur les approches proposées en ce qui concerne l'étude de la prédisposition des composants logiciels aux changements et aux bogues.

Ensuite, nous avons réalisé l'étude empirique afin d'analyser l'impact de différents types de couplage sur la prédisposition des clients aux changements. Pour répondre à nos hypothèses de recherche, nous avons suivi une démarche de collecte de données basée sur deux étapes. La première consiste à identifier les classes clients stables ayant seulement des couplages avec les

interfaces et les clients instables ayant des couplages avec les implémentations. La deuxième étape repose sur la détection des changements des clients identifiés et leur implication dans les bogues. Dans le contexte de l'étude, nous avons choisi d'analyser trois systèmes développés en Java. Nous avons aussi utilisé des méthodes d'analyse statistique pour vérifier la validité de nos hypothèses.

En dernier lieu, nous avons présenté de façon objective les résultats obtenus à partir de l'application des tests statistiques sur les données collectées. Ensuite, nous avons discuté de façon subjective les résultats présentés selon des cas concrets dans les systèmes étudiés.

Organisation du mémoire

Le reste du mémoire est organisé comme suit : le chapitre 1 présente une revue de la littérature sur les travaux liés à notre contribution. Le chapitre 2 décrit l'étude empirique effectuée concernant la collecte de données, les hypothèses et les variables de recherche, ainsi que les méthodes d'analyse statistique appliquées. Le chapitre 3 comporte la présentation et la discussion des résultats statistiques obtenus, les obstacles à la validité et les limites de l'approche. La conclusion générale et les perspectives du projet sont présentées dans le dernier chapitre.

CHAPITRE 1

REVUE DE LA LITTÉRATURE

Les interfaces sont des mécanismes importants dans la conception et la programmation orientée objet. Elles constituent des abstractions permettant de cacher les parties complexes qui sont constamment exposées aux variations. Cette dissimulation protège les entités externes des changements dans la conception des détails et favorise le faible couplage. Cependant, les mauvaises utilisations de ce patron peuvent avoir un impact sur la stabilité des artefacts logiciels en ce qui concerne leur prédisposition aux changements propagés par les variations qui doivent être cachées derrière l'interface. Ces pratiques peuvent aussi augmenter les coûts de maintenance et mener à la dégradation des systèmes.

Dans ce chapitre, nous présentons les principaux travaux effectués qui sont liés à notre sujet. Nous commençons par évoquer les différentes définitions accordées à la stabilité dans la littérature, les métriques de mesure de la stabilité des composants logiciels et les principes de conception proposés par les chercheurs pour favoriser la stabilité des conceptions. Ensuite, nous présentons un survol sur le mécanisme d'interface dans la programmation orientée objet et les différentes techniques promouvant l'utilisation efficace de cette technique. Dans la dernière section, nous abordons les approches proposées sur l'étude de la prédisposition des composants logiciels aux changements et aux bogues.

1.1 Stabilité des logiciels

Dans la littérature, plusieurs définitions ont été accordées à la stabilité des logiciels. Selon Hassan (2007) et Alshayeb *et al.* (2011), ces définitions se répartissent en trois vues générales. La première approche définit la stabilité comme la résistance aux changements apportés au système. Le logiciel est stable lorsque deux versions consécutives sont quasiment similaires. Cette dernière a été proposée par Martin (1997) et Soong (1977).

La deuxième définition adoptée par Yau et Collofello (1980), Elish et Rine (2003) et Fayad et Altman (2001), indique qu'un système est stable lorsque ce dernier est capable de résister à la propagation des changements dans le cas d'ajout de nouveaux artefacts ou l'amélioration du code existant. Une classe est dite stable lorsqu'elle subit des modifications sans affecter le reste du système. Selon McConnell (2004), l'accommodation des changements dans une conception a pour objectif d'isoler les composants instables afin de limiter l'effet de leur changement. Dans le même contexte, la norme ISO-1926 (2001) décrit la stabilité des logiciels comme la sensibilité à apporter des changements sur un système et qui est un impact négatif dû à un autre changement. La troisième définition considère un système comme stable lorsqu'il n'y a aucun ajout de nouveaux composants ou modification du code existant, ou lorsqu'il y a seulement l'ajout de nouvelles fonctionnalités et sans faire des changements (Grosser *et al.* 2002 ;Jazayeri 2002).

D'autre part, les chercheurs ont proposé différentes approches basées sur des propriétés mesurables afin de faciliter l'analyse de la stabilité des conceptions et de réduire les efforts consacrés à la maintenance. Kelly (2006) a proposé une définition de la stabilité basée sur les mesures des caractéristiques (par exemple, le nombre de lignes du code source d'un module) de la conception. L'auteur a considéré qu'une caractéristique est stable lorsque les valeurs des métriques associées à cette dernière sont presque similaires entre les versions des logiciels.

1.1.1 Métriques de mesure de la stabilité

Les métriques logicielles sont des mesures quantitatives permettant d'analyser des propriétés spécifiques telles que le couplage, la cohésion, etc. Il existe dans la littérature différentes métriques suggérées pour mesurer la stabilité des architectures et des composants logiciels. Yau et Collofello (1980) ont proposé des mesures pour la stabilité de la conception et du code source des composants logiciels en ce qui concerne l'impact de leur modification sur le reste du système. Les métriques proposées ont considéré la complexité des modules logiciels selon leur degré de dépendances, la probabilité de leur modification et la portée des variables utilisées. Dans un même contexte, Arvanitou *et al.* (2015) ont proposé la métrique REM (*Ripple Effect*

Measure) qui consiste à évaluer la prédisposition d'une classe logicielle aux changements propagés par d'autres classes. Martin (1997) a proposé des métriques de calcul de la stabilité liée aux architectures des logiciels. L'auteur a considéré les dépendances entre les paquetages dans la programmation orientée objet. Les mesures constituent les couplages afférents et efférents entre les modules, et le degré d'instabilité selon les deux types de couplage. Bansiya (2000) a proposé une série de mesures permettant d'évaluer la stabilité architecturale des cadriciels dont la programmation est orientée objet. L'auteur a considéré les caractéristiques structurelles liées au niveau des classes logicielles telles que le nombre des classes dans le système, le nombre des hiérarchies des classes, la moyenne de la profondeur de l'héritage dans la conception, etc. Tonu *et al.* (2006) ont présenté une approche basée sur des métriques afin d'évaluer la stabilité architecturale et dans quelle mesure une architecture logicielle peut supporter les changements. Les auteurs ont adopté une analyse rétrospective et prédictive dans leur expérimentation. Ils ont identifié que la complexité, la cohésion et le couplage sont des facteurs de contribution dans la stabilité des architectures.

Différentes métriques ont été aussi suggérées afin de mesurer le degré de la stabilité des classes logicielles dans leur évolution. Li *et al.* (2000) ont proposé une mesure de l'instabilité d'une classe entre deux versions d'un système. La métrique concerne le pourcentage du changement effectué dans le code source des classes. Grosser *et al.* (2003) ont pris en considération la stabilité d'une classe selon le changement au niveau des méthodes de l'interface qu'elle implémente. Rapu *et al.* (2004) ont proposé de mesurer la stabilité des classes selon les méthodes ajoutées ou supprimées. Les auteurs ont considéré une classe stable lorsque le nombre de méthodes ne change pas entre les versions d'un logiciel. Alshayeb *et al.* (2011) ont identifié une liste de propriétés des classes qui peuvent affecter leur stabilité. Ils ont considéré les changements dans les variables et leurs modificateurs, les signatures des méthodes, les noms des interfaces et des classes héritées.

Les mesures quantitatives sont des techniques d'évaluation définies afin de calculer le degré de la stabilité des artefacts logiciels et de déterminer si un composant sera exposé aux changements dans un sens prédictif. En outre, les chercheurs ont proposé des mécanismes et des

principes de conception promouvant plus de stabilité des structures logicielles et la réduction de l'impact des changements dans l'évolution des systèmes.

1.1.2 Principes de conception liés à la stabilité des logiciels

Dans la littérature, plusieurs principes de conception ont été proposés afin de promouvoir la stabilité logicielle et la capacité à gérer les changements au fil du temps. Parnas (1972) a proposé le mécanisme de masquage d'information qui consiste en la création des modules permettant de cacher les décisions de conception qui sont difficiles ou présentent des possibilités de changement. Il a constaté des conflits difficiles entre le besoin de spécifier (et communiquer les détails) des aspects d'un système assez tôt dans son développement et le besoin de pouvoir modifier des aspects du même système plus tard à cause des changements inévitables. Il a proposé donc une heuristique de ne pas communiquer (documenter) les informations sur les détails difficiles ou instables. Tout ce qui était masqué pouvait être changé. Ce concept favorise la stabilité des conceptions en dissimulant les composants logiciels susceptibles d'être modifiés afin de réduire l'impact de leurs changements sur les autres modules. Le masquage d'information a inspiré les approches de la programmation modulaire dans la création des systèmes complexes. De plus, il a permis aux développeurs de travailler en parallèle tout en diminuant les dépendances entre les modules et en simplifiant la coordination nécessaire.

Dans un contexte similaire, Meyer (1988) a suggéré le principe ouvert-fermé dans la programmation orientée objet. Les modules doivent être ouverts aux extensions et fermés aux modifications. Ceci veut dire qu'une fois le code source d'un module a été compilé, les nouvelles fonctionnalités peuvent être ajoutées par extension (par exemple, l'ajout d'une implémentation ou d'une méthode), ou à travers la réutilisation des modules existants. L'utilité de l'application de ce mécanisme dans une conception est d'éviter l'introduction des bogues dans le système et de maintenir la stabilité des entités logicielles qui utilisent les fonctionnalités d'un composant fermé au changement.

La troisième idée similaire au masquage d'information est la protection des variations (Cockburn 1996 ; Larman 2001). Ce concept s'appuie sur la dissimulation des éléments prédisposés aux changements derrière une interface stable. L'introduction d'une frontière entre les points de variations et les clients (un composant logiciel qui invoque ou utilise une fonctionnalité implémentée dans un autre composant) minimise le couplage envers les éléments instables, le coût de changement et facilite l'ajout de nouvelles fonctionnalités sans affecter les clients. La protection des variations est la principale racine ayant motivé la plupart des mécanismes (par exemple, les interfaces, l'encapsulation de données) et des patrons (plusieurs patrons GoF) dans la programmation et la conception supportant la flexibilité (Larman, 2001).

Les patrons de conception GoF (Gamma *et al.*, 1994) constituent des solutions pouvant être utilisées pour résoudre des problèmes courants dans la conception orientée objet. L'utilisation de ces patrons offre plusieurs avantages tels que l'amélioration de la réutilisabilité, la maintenabilité et l'adaptabilité des systèmes. Depuis leur apparition, les patrons GoF ont été largement étudiés dans la communauté de recherche en génie logiciel. L'étude de Ampatzoglou *et al.* (2013) a révélé que les chercheurs ont montré plus d'intérêt au sujet de l'impact des patrons GoF sur les attributs de qualité et plus spécifiquement sur l'impact de l'application des patrons sur la maintenabilité et la stabilité des logiciels. Une première contribution sur le lien entre la maintenance et les patrons a été effectuée par Prechelt *et al.* (2001). Les auteurs ont réalisé une étude expérimentale dont l'objectif est de comparer une application implémentée en utilisant des patrons GoF avec une solution simple de la même application. L'étude a été menée sur les patrons Fabrique abstraite, Décorateur, Visiteur, Composite et Observateur. Des tâches de maintenance (amélioration de la conception) ont été proposées à des professionnels afin de mesurer leur durée et leur degré de complexité pour les deux applications. Les résultats suggèrent qu'il est parfois préférable d'utiliser les patrons au lieu des simples conceptions. Cette étude a été répliquée par Vokáč *et al.* (2004) en utilisant les mêmes patrons dans un environnement de programmation réel. Les résultats concluent que l'avantage de l'application d'un patron dépend de sa nature et le contexte de son utilisation. Les auteurs trouvent que le patron Observateur possède un effet positif, alors que Visiteur est difficile à comprendre et à maintenir. Cependant,

d'autres réplications (Juristo et Vegas 2011 ; Nanthaamornphong et Carver 2011 ; Krein *et al.* 2011) de l'étude de Prechelt *et al.* (2001) ont produit des résultats contradictoires ne permettant pas d'identifier l'impact réel d'un patron sur la maintenabilité des logiciels. De plus, des outils ont été proposés afin d'évaluer l'avantage d'existence d'un patron sur la maintenance des systèmes (Ampatzoglou *et al.* 2012 ; Alghamdi et Qureshi 2014). Toutefois, les solutions proposées ne traitent pas la majorité des patrons de conception GoF et les techniques de leur évaluation sont limitées.

Les chercheurs ont aussi montré leur intérêt à étudier l'impact des patrons de conception GoF sur la stabilité des logiciels. Elish (2006) a évalué l'impact de quatre patrons structurels GoF sur la stabilité de la conception et plus précisément sur la résistance à la propagation des changements. L'approche suggère que les patrons Adaptateur, Composite, Façade et Pont donnent davantage stabilité en ce qui concerne l'évolution de diagramme de classes. Ampatzoglou *et al.* (2015) ont récemment effectué une étude sur la stabilité des patrons GoF dans l'évolution des conceptions logicielles. Les auteurs ont adopté la définition de la stabilité décrite dans la norme ISO-1926 (2001) et qui porte sur la résistance des systèmes à la propagation des changements. L'étude analyse la fréquence de changements des classes d'un patron et qui sont propagés par d'autres modifications. Les résultats présentés dans ce travail montrent que les classes ayant un rôle dans un seul patron sont plus stables que celles associées à plusieurs patrons. De plus, les auteurs concluent que le niveau de stabilité dépend du type de patron. Par exemple, les classes appartenant aux patrons Observateur, Façade, Composite et Décorateur sont plus stables dans les systèmes étudiés. Toutefois, cette approche se limite seulement à la propagation des changements entre les composants logiciels. D'autres études ont porté sur tout type de changement (par exemple, l'ajout de nouvelles fonctionnalités, la correction des bogues) pouvant être effectué sur les classes des patrons. Bieman *et al.* (2001) ont proposé une étude industrielle pour analyser la corrélation entre la fréquence de changement des classes appartenant aux patrons GoF et leur taille. Les auteurs ont trouvé que le nombre de modifications est fortement corrélé avec la taille d'une classe lorsque cette dernière appartient à un patron GoF. Aversano *et al.* (2007) ont présenté une étude sur l'évolution des patrons de conception GoF dans trois logi-

ciels implémentés en Java. Les auteurs ont analysé la fréquence de changement des patrons et les types de changement effectués par les développeurs. Les résultats indiquent que les patrons étudiés sont souvent modifiés dans les cas d'ajout de nouvelles sous-classes, la modification des implémentations des patrons ou les méthodes des interfaces. Toutefois, la fréquence de changement des patrons ne dépend pas du type, mais du contexte de son application et de la fonctionnalité supportée. Gatrell *et al.* (2009) ont réalisé une étude empirique sur la prédisposition des patrons GoF aux changements dans l'évolution d'un logiciel commercial développé en C#. Les auteurs ont constaté une corrélation considérable entre les classes des patrons Adaptateur, Proxy, Singleton, Stratégie et État, et leur degré de changement.

1.2 Mécanisme d'interface en OO

Selon McConnell (2004), la première et la plus importante étape de la création des classes de haute qualité est l'introduction de bonne interface. Cette dernière consiste à créer une abstraction à travers l'interface afin d'assurer le masquage des détails. Ce patron est largement utilisé dans la programmation orientée objet et dans la création des librairies logicielles. Il offre une solution permettant de cacher les implémentations par rapport aux clients de l'interface afin de minimiser le couplage et de réduire l'impact de changement des artefacts instables. Une interface déclare un ensemble de contrats de service qui définissent la façon d'interaction entre les modules logiciels. Romano et Pinzger (2011) a mentionné que ces contrats doivent être stables dans l'évolution des logiciels afin de réduire l'effort de la compréhension et de la maintenance des systèmes. Cependant, la conception de bonnes interfaces n'est pas une tâche triviale (Henning, 2007), et possède une grande influence sur le reste du système. Les défauts de conception dans les interfaces peuvent mener à l'augmentation des coûts de développement, et à la dégradation de la qualité du code.

Dans la littérature, le concept d'interface a été défini comme une spécialisation du principe de masquage d'information proposé par Parnas (1972). D'ailleurs, Larman (2005) a indiqué que les interfaces sont des mécanismes de protection des variations. Lors de leur conception,

le développeur décide les fonctionnalités qui seront visibles à l'extérieur et qui forment des abstractions pour les implémentations cachées derrière l'interface.

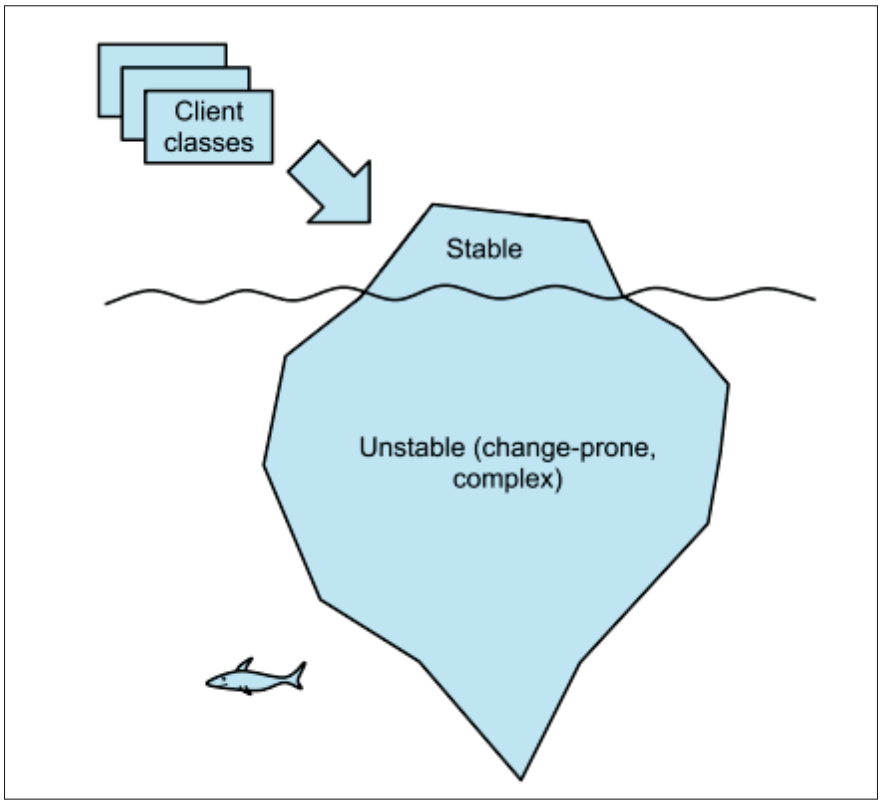

Figure 1.1 Iceberg expliquant le principe de masquage d'information proposé par McConnell (2004) et adapté par Rufiange et Fuhrman (2014).

McConnell (2004) a présenté le principe de masquage d'information sous forme d'un iceberg (Figure 1.1). Les implémentations d'une interface stable constituent la partie complexe qui est susceptible aux changements. Les classes clients sont capables d'appeler seulement les abstractions révélées par l'interface. L'isolation des éléments qui risquent d'être modifiés donne davantage de stabilité aux clients et facilite l'ajout de nouvelles extensions. Les interfaces favorisent aussi la réutilisabilité du code à travers le polymorphisme, le faible couplage et la facilité de la maintenance des systèmes.

Il existe une variété de techniques de conception permettant une meilleure utilisation du mécanisme d'interface. Martin (2000) a proposé le concept de la ségrégation d'interface (*Interface segregation principle*) qui consiste à créer des interfaces plus spécifiques au lieu de l'utilisation d'une interface générale contenant toutes les fonctionnalités. Ce concept favorise la cohésion des implémentations lorsque ces dernières dépendent seulement des interfaces dont elles ont besoin de définir leurs contrats.

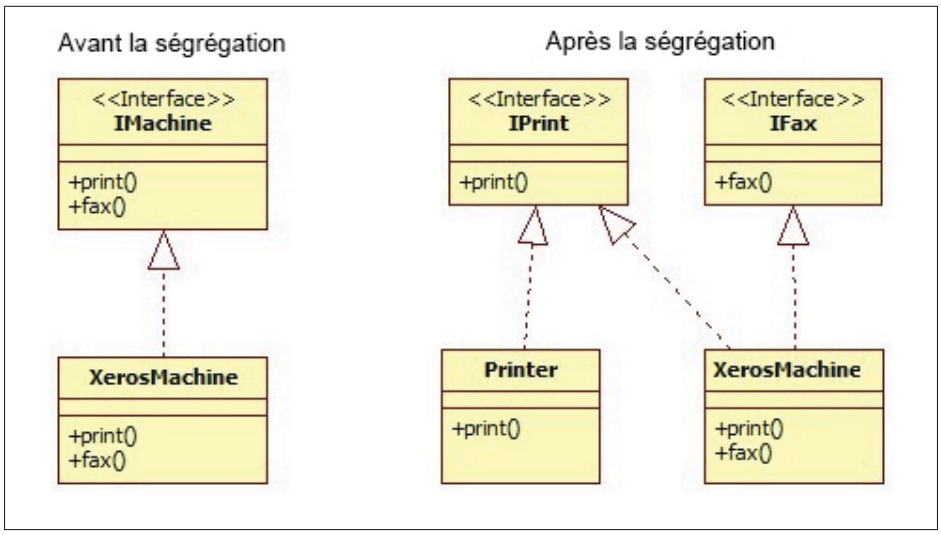

Figure 1.2 Exemple du principe de la ségrégation d'interface tiré de Martin (2000).

La Figure 1.2 présente un exemple de la ségrégation d'une interface. La classe *XerosMachine* définit les fonctionnalités *print* et *fax* déclarées dans l'interface *IMachine*. Toutefois, la classe *Printer* a besoin seulement d'implémenter la méthode *print*. Dans ce cas, l'interface *IMachine* doit être ségrégée afin de ne pas forcer la classe *Printer* à redéfinir la méthode *fax*. Selon Romano et Pinzger (2011), la violation de ce principe dans la conception des interfaces est associée à un risque de changement. Les auteurs ont utilisé deux métriques de cohésion afin de mesurer cette propriété dans l'évolution des logiciels. Ils ont trouvé que les mesures effectuées montrent une corrélation entre le manque de cohésion des interfaces et leur prédisposition aux changements.

Gamma *et al.* (1994) ont proposé le principe de la programmation à l'interface et non aux implémentations. L'idée relative à ce principe est de réduire le couplage des clients aux implémentations de l'interface afin d'augmenter la réutilisabilité et la flexibilité des systèmes, et d'éviter la propagation du changement des composants instables cachés derrière l'interface. Dans un contexte similaire, Fowler (2004) a décrit le patron inversion de contrôle comme une solution permettant de découpler le code client de la création des objets dérivés d'une interface. Ce concept facilite les tests unitaires et favorise le faible couplage. L'injection de dépendance a été proposée par Fowler (2004) comme une implémentation de l'inversion de contrôle afin de résoudre le problème des dépendances. Ce mécanisme s'appuie sur l'injection des objets à travers des techniques de création d'instances telles que les patrons de fabrication ou les fichiers de configuration (l'élément injecteur dans la Figure 1.3).

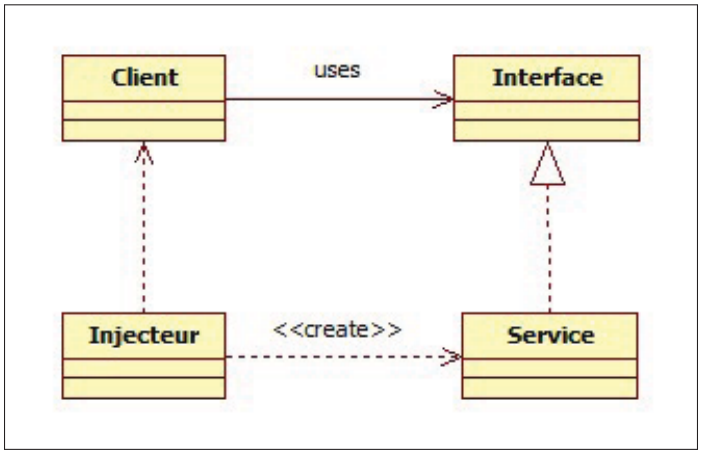

Figure 1.3 Implémentation de l'inversion de contrôle avec l'injection de dépendance proposée par Fowler (2004).

L'injection peut être effectuée via le constructeur des clients, une méthode setter déclarée dans le code client ou à travers l'implémentation des interfaces déclarant des contrats d'injection. L'élément injecteur injecte des références sur les objets afin d'éviter les couplages création-

nelles au niveau des clients. L'inversion de contrôle a inspiré plusieurs cadriciels de développement tels que Google Guice [1] (Vanbrabant, 2008) ou Spring [2] (Fowler, 2004).

D'autres travaux ont été proposés afin d'étudier l'impact des défauts de conception des interfaces sur la qualité des logiciels. Boxall et Araban (2004) ont défini un ensemble de métriques permettant de mesurer la complexité des interfaces dans les systèmes. Leurs métriques calculent le nombre des méthodes et des clients dans une interface, et le nombre des arguments dans les méthodes. Romano et Pinzger (2011) ont proposé une métrique de mesure relative au degré de la violation du principe de la ségrégation des interfaces. Abdeen *et al.* (2013a) ont défini une métrique de faible programmation à l'interface. Cette mesure permet d'évaluer la conformité des interfaces au principe de la programmation à l'interface et non aux implémentations proposé par Gamma *et al.* (1994). Cataldo *et al.* (2010) ont examiné l'impact de la complexité des interfaces sur l'introduction des bogues en se basant sur les métriques proposées par Bandi *et al.* (2003) et qui mesurent la taille des interfaces et la complexité des arguments des méthodes déclarées. Les auteurs ont constaté que la complexité élevée des interfaces a un impact négatif en ce qui concerne l'exposition des implémentations aux changements. Abdeen *et al.* (2013b) ont réalisé une étude empirique sur l'impact de deux anomalies de conception des interfaces (la déclaration des interfaces avec des méthodes dupliquées ; l'existence des méthodes non utilisées dans les interfaces) sur la maintenabilité et la complexité des systèmes. Les auteurs ont trouvé que les défauts de conception liés aux interfaces se trouvent dans la majorité des logiciels étudiés. Les développeurs déclarent souvent des méthodes non utilisées ou dupliquées. Ces pratiques affectent la cohésion et la taille d'une interface et peuvent violer le principe de la ségrégation des interfaces.

1.2.1 Modèle d'une zone de variabilité

Dans (Rufiange et Fuhrman, 2014), les auteurs ont proposé le concept d'une zone de variabilité qui illustre les éléments clés d'une interface Java (Figure 1.4). La protection des variations de

1. http://github.com/google/guice
2. http://spring.io

l'interface ne peut être achevée que si les clients sont couplés uniquement avec l'interface. Les clients ayant du couplage avec les implémentations peuvent être exposés aux changements et aux bogues.

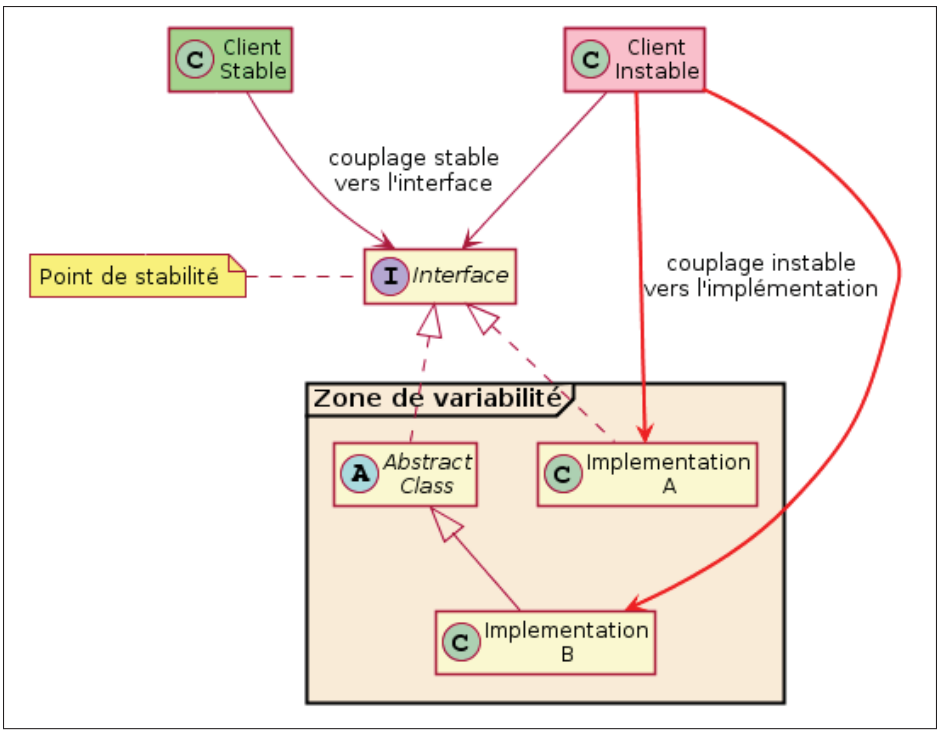

Figure 1.4 Modèle d'une zone de variabilité proposé par Rufiange et Fuhrman (2014).

Le modèle proposé définit une interface comme un point de stabilité qui constitue une indirection entre les clients et la zone de variabilité. Cette dernière comporte principalement les implémentations de l'interface qui peuvent être des classes abstraites, concrètes ou les interfaces qui étendent le point de stabilité.

Un client est dit stable lorsque ce dernier a seulement du couplage avec l'interface. Ce type du client est censé être protégé des changements qui peuvent affecter la zone de variabilité. Plusieurs techniques de conception ont été proposées et qui favorisent le couplage stable des clients de l'interface (programmation à l'interface et non à l'implémentation (Gamma *et al.*,

1994) ; inversion de contrôle (Fowler, 2004)). Les clients stables sont privés de l'instanciation des objets dérivés de l'interface. Cette responsabilité peut être affectée à un patron de création tel que les fabriques dans GoF, ou à travers l'injection de dépendances.

Les clients instables sont les classes qui dépendent simultanément de l'interface et de ses implémentations. Ce type de couplage est jugé coûteux (Izurieta et Bieman, 2013), et implique que les clients ne sont pas protégés aux changements qui se produisent à l'intérieur de la zone de la variabilité. Les dépendances instables peuvent aussi introduire les bogues dans le système et augmentent le nombre du couplage risqué.

L'approche de Rufiange et Fuhrman (2014) constitue principalement un prototype de visualisation de l'évolution des zones de variabilité. Cette contribution permet également d'étudier la protection des variations des interfaces en identifiant les clients instables et les clients stables. Toutefois, il existe un manque d'études empiriques sur l'impact de l'existence des couplages avec les implémentations en ce qui concerne la stabilité des clients des interfaces dans l'évolution des logiciels.

1.3 Prédisposition des systèmes aux changements

Les changements apportés à un système sont nécessaires dans l'évolution des logiciels afin de préserver leur utilité. Ils peuvent constituer les ajouts de nouvelles fonctionnalités, les modifications des composants existants, les corrections des défauts, l'adaptation du logiciel dans un nouvel environnement, etc. Toutefois, la fréquence du changement peut varier selon la nature et le rôle des artefacts logiciels, et peut aussi affecter le degré de leur stabilité dans l'évolution.

En programmation orientée objet, la prédisposition aux changements (*change-proneness*) est une mesure utilisée pour calculer le nombre de modifications des classes dans l'histoire des logiciels. Selon Jaafar *et al.* (2014a) et Ampatzoglou *et al.* (2015), cette mesure concerne tout type de changement qui se produit dans une classe (par exemple, introduction de nouvelles fonctionnalités, amélioration du code existant, correction de bogues). Les changements peuvent être extraits à partir des systèmes de gestion des versions tels que CVS, SVN ou Git. Gall *et al.*

(2003) ont considéré un changement d'une classe dans une révision comme une action d'ajout, de modification ou de suppression des lignes du code source. Cette définition a été adoptée par Di Penta *et al.* (2008) et Aversano *et al.* (2007) dans leurs études sur la prédisposition des patrons GoF aux changements.

Le calcul du nombre des changements dans l'évolution des logiciels est un facteur très utile pour distinguer entre les artefacts qui changent fréquemment, et ceux qui sont moins sujets aux modifications. Dans un processus de maintenance, des mesures de corrections ou d'améliorations spécifiques peuvent être prises en considération pour les composants logiciels les plus exposés aux changements. De plus, l'étude de l'évolution donne davantage d'idées sur les différents types de changements (par exemple, ajout, modification, suppression de méthode dans une classe) les plus effectués par les développeurs, et aussi sur la fréquence de changements selon les rôles joués par les classes dans la conception (par exemple, degré de changement selon le rôle d'une classe dans un patron de conception). L'étude peut aussi révéler les défauts de conception à travers les changements effectués dans un contexte de correction de bogues. Les classes qui participent souvent dans des actions correctives se dégradent et peuvent devenir obsolètes. Les données issues de l'historique des changements dans les dépôts des logiciels libres ont permis aux chercheurs d'analyser empiriquement l'évolution des entités et des structures de conceptions tels que les patrons GoF de Gamma *et al.* (1994).

1.3.1 Prédisposition des patrons Gang Of Four aux changements

Dans la littérature, plusieurs études empiriques ont été réalisées sur la prédisposition des patrons de conception GoF aux changements. Bieman *et al.* (2001) ont étudié l'évolution des systèmes orientés objet afin d'analyser la corrélation entre les patrons de conception GoF et leur prédisposition aux changements. Les auteurs ont trouvé que les classes qui participent dans un patron GoF et les classes avec un grand nombre de méthodes sont plus exposées aux changements dans l'évolution des logiciels. De plus, les classes réutilisées à travers l'héritage peuvent introduire la propagation des changements aux sous-classes. Toutefois, cette étude n'a pas considéré les différents rôles joués par les classes (par exemple, les classes clients des pa-

trons) dans les structures des patrons GoF, les types des changements les plus effectués par les développeurs, et l'évolution entre les révisions et non pas entre les versions. De plus, l'identification des patrons GoF dans les projets étudiés s'est basée seulement sur la documentation. Cette étude a été répliquée par les mêmes auteurs dans (Bieman *et al.*, 2003). Ils ont analysé cinq projets avec les mêmes objectifs de recherche. Cependant, ils ont trouvé que la corrélation entre la taille d'une classe et le nombre de changements n'est pas totalement supportée dans les systèmes étudiés. Aversano *et al.* (2007) ont réalisé une étude empirique sur la prédisposition des patrons GoF aux changements, les types de modification effectués et leur impact sur les classes clients des patrons. L'étude a été effectuée sur trois systèmes à code source ouvert en considérant l'évolution entre les révisions. Les auteurs ont utilisé l'outil de Tsantalis *et al.* (2006) pour la détection des patrons dans les projets étudiés. Les résultats ont montré que les classes des patrons changent fréquemment, et que les changements ne dépendent pas du type d'un patron, mais aussi du contexte de son application et de la fonctionnalité supportée. Toutefois, les auteurs n'ont pas trouvé un résultat significatif en ce qui concerne le lien entre les changements dans les patrons et leur impact sur les classes clients parce qu'ils ont considéré seulement la propagation d'un changement au lieu des autres types de changements qui peuvent se produire dans une classe. Les clients jouent un rôle important dans la structure de la plupart des patrons GoF. Cependant, cette étude a porté sur la prédisposition des classes des patrons aux changements d'une façon générale alors qu'une structure d'un patron GoF est composée de plusieurs rôles. L'étude de Aversano *et al.* (2007) a été répliquée par Di Penta *et al.* (2008), mais en étudiant également les changements selon les rôles joués par les classes des patrons. Les auteurs ont utilisé l'outil DeMiMa de Guéhéneuc et Antoniol (2008) qui permet d'identifier dans un système Java les patrons GoF implémentés, ainsi que les différents rôles selon le nom du patron. Les résultats trouvés indiquent que le nombre de changements varie selon les rôles dans une structure d'un patron de conception GoF. Par exemple, dans le cas du patron Adaptateur, les classes qui jouent le rôle d'un adaptateur sont plus exposées aux changements que les classes adaptées. De plus, les fabriques abstraites sont moins modifiées que les fabriques concrètes, ainsi que les classes composites peuvent devenir plus complexes dans l'évolution du patron Composite. Les auteurs ont aussi trouvé que les modifications au

niveau des méthodes des interfaces peuvent faire d'autres parties du système moins robuste aux changements. Toutefois, l'étude menée par Di Penta *et al.* (2008) n'a pas considéré les défauts de conception à travers les changements effectués dans les révisions de correction de bogues. La révélation des rôles les plus impliqués dans ce type de changement confirme l'importance de la bonne conception de ces classes. Dans (Khomh *et al.*, 2009b), les auteurs ont étudié la corrélation entre le nombre de patrons auxquels une classe appartient, et sa prédisposition aux changements dans l'évolution des logiciels. Les résultats ont révélé que les classes qui jouent plusieurs rôles sont plus exposées aux modifications, possèdent plus de couplage et sont moins cohérentes que les classes ayant un seul rôle. Finalement, Ampatzoglou *et al.* (2015) ont proposé une étude de l'impact des patrons GoF sur la stabilité en ce qui concerne la propagation des changements. Les auteurs ont analysé la différence de la résistance aux changements des classes qui participent dans au moins un patron GoF, et les classes qui n'appartiennent à aucun patron. Ils ont aussi étudié la différence de la stabilité des classes selon le nombre des patrons dans lesquels elles jouent un rôle. Les résultats de cette approche ont montré que les classes qui jouent exactement un seul rôle sont plus stables que celles qui ont zéro ou plusieurs rôles. De plus, les auteurs ont trouvé que la résistance à la propagation des changements dépend des rôles joués par les classes des patrons étudiés. Par exemple, les classes qui possèdent un rôle dans les patrons Observateur, Composite, et Décorateur sont moins exposées aux changements propagés par d'autres modifications. Cependant, Ampatzoglou *et al.* (2015) ont souligné que dans 10413 classes identifiées dans les patrons GoF, il existe peu de classes (seulement 156 clients) qui jouent le rôle d'un client. Cette lacune est due à la limitation de l'outil de détection des patrons et les rôles des classes.

Dans l'évolution des logiciels, il est important de distinguer entre les composants stables et les composants qui sont souvent modifiés par les développeurs. Les classes les plus prédisposées aux changements exigent davantage d'efforts consacrés à leur maintenance. L'étude de l'histoire des changements dans l'évolution des logiciels a donné aux chercheurs de nouvelles opportunités telles que les approches de la prédiction des changements ou la prédisposition des anti-patrons aux changements.

1.3.2 Prédisposition des anti-patrons aux changements

En génie logiciel, les anti-patrons sont des mauvaises décisions de conception qui engendrent souvent un coût de maintenance élevé. Brown *et al.* (1998) ont proposé une liste de 40 anti-patrons qui sont des pratiques coûteuses et qui exposent les systèmes aux changements et à l'introduction des bogues.

Dans la littérature, plusieurs travaux ont été réalisés sur la prédisposition aux changements des classes implémentant des anti-patrons. Khomh *et al.* (2009b) ont proposé une étude empirique sur le lien entre l'existence des anti-patrons dans des classes logicielles et leur prédisposition aux changements. Ils ont analysé l'impact du nombre des anti-patrons et leurs types sur le degré de modification des classes dans l'évolution des logiciels. Les auteurs ont trouvé que les classes avec des anti-patrons sont plus exposées aux changements que les autres classes. De plus, il existe une corrélation entre le nombre d'anti-patrons qui se trouvent dans une classe et sa fréquence de changements. Olbrich *et al.* (2009) ont analysé l'historique de l'évolution des anti-patrons *God Class* [3] et *Shotgun Surgery* [4] dans les projets Xerces et Lucence. Les auteurs ont conclu que les classes affectées par ces deux anti-patrons ont un degré de changement plus élevé par rapport aux autres classes. Romano *et al.* (2012) ont étudié l'impact des anti-patrons sur le changement dans le code source des classes logicielles. Ils ont analysé les types de changement effectués par les développeurs tels que les changements fonctionnels (ajout, suppression de méthodes), les changements dans les structures conditionnelles, les changements dans les attributs de la classe et les changements dans l'API (les changements qui affectent la déclaration de la classe, les signatures des méthodes, etc). Ce dernier type a été considéré comme le changement le plus effectué dans les classes avec des anti-patrons. De plus, les auteurs ont constaté que le type de changement et sa fréquence varient selon différents anti-patrons.

3. un objet peu cohésif qui contient plusieurs fonctionnalités
4. obligation de modification de plusieurs classes à cause d'un changement donné

1.4 Prédisposition des systèmes aux bogues

La prédisposition aux bogues (*fault-proneness)* d'un composant logiciel désigne principalement sa fréquence de changements due aux corrections des défauts de conception. Un bogue dans un logiciel représente une source de dysfonctionnement qui peut affecter l'exécution normale du système. Les bogues informatiques sont souvent documentés par les développeurs dans des logiciels de suivi de problèmes tels que Bugzilla [5] ou Jira [6]. Ces logiciels sont utilisés pour signaler les anomalies identifiées dans le code source et les tâches effectuées pour les corriger.

Vokáč (2004) a proposé une étude empirique sur la prédisposition aux bogues des cinq patrons GoF : Singleton, Méthode Template, Décorateur, Observateur, Méthode Fabrique. Son analyse a porté sur l'évolution d'un système commercial développé en langage C++. L'auteur a trouvé une forte corrélation entre l'augmentation de la taille des classes des patrons Observateur et Singleton, et leur fréquence de changement dans des révisions de correction des bogues. De plus, le patron méthode fabrique possède moins de couplage et il est moins exposé aux bogues. Par contre, Vokáč (2004) n'a pas trouvé des résultats significatifs pour le reste des patrons étudiés. Cependant, cette étude n'a pas pris en considération les rôles des classes dans les structures des patrons et les types des changements effectués par les développeurs dans les révisions de corrections des bogues. Ce genre d'analyse donne plus d'aperçus sur la différence entre les degrés de modifications des rôles afin de distinguer les classes qui nécessitent plus d'attention dans leur conception. Dans (Gatrell et Counsell, 2011), les auteurs ont analysé la relation entre les classes participantes dans 13 patrons GoF et leur prédisposition aux bogues dans l'évolution d'un système commercial développé en C#. Ils ont trouvé des résultats positifs en ce qui concerne les classes dans les patrons étudiés et leur implication dans des corrections de bogues. Les patrons Observateur, Singleton et Méthode Template sont plus exposés à ce type de changement. Ampatzoglou *et al.* (2011) ont réalisé une étude empirique sur la fréquence de correction des défauts relatifs à une liste de 11 patrons GoF implémentés dans des jeux

5. http://www.bugzilla.org
6. http://jira.atlassian.com

développés en Java. Les auteurs ont observé que les patrons Adaptateur et Méthode Template sont positivement corrélés avec le degré de modification dans un cadre de correction de bogues.

Les patrons de conception GoF sont souvent cités comme des solutions avantageuses pour la réutilisabilité, la maintenabilité et la résistance aux changements dans l'évolution des logiciels. Toutefois, les chercheurs ont trouvé dans des cas réels que ces patrons peuvent parfois introduire de la complexité au système et favoriser l'augmentation du nombre de changements et le nombre de corrections de bogues.

CHAPITRE 2

ÉTUDE EMPIRIQUE

Nous nous intéressons dans ce travail à étudier l'impact de l'existence du couplage instable sur la qualité des logiciels. À partir d'un ensemble de projets réels, nous cherchons à déterminer si les dépendances avec les implémentations des interfaces favorisent la prédisposition des clients aux changements et aux bogues. L'étude va révéler plus d'aperçus sur la conséquence de ce type de couplage en ce qui concerne la stabilité des clients dans l'évolution des logiciels.

Cette étude porte sur l'application d'une méthode de validation empirique dont l'objectif principal est d'explorer l'utilité des concepts théoriques dans la pratique. Nous proposons deux questions de recherche liées à notre problématique. Nous effectuons aussi un processus de collecte des données à partir d'une liste de projets sélectionnés selon des critères spécifiques. Enfin, nous réalisons une analyse statistique des résultats obtenus pour vérifier la validité de nos hypothèses de recherche.

Dans ce chapitre, nous présentons la méthode de recherche adoptée dans notre étude. Nous commençons par présenter les outils et les étapes suivies pour la collecte des données. Ensuite, nous présentons les questions et les hypothèses de recherche proposées, les variables de recherche et le contexte de l'étude empirique. La dernière section décrit les méthodes d'analyse statistique utilisées dans cette étude.

2.1 Extraction et collecte des données

La méthodologie adoptée dans cette étude a été basée sur des données collectées selon un processus comportant deux étapes (Figure 2.1). La première consiste à générer le modèle de stabilité y compris la liste des clients stables et des clients instables à partir du dépôt du code source. La deuxième étape consiste à calculer le nombre de changements et de bogues de chaque client identifié dans l'étape précédente, ainsi que l'analyse statistique des résultats obtenus. De plus,

nous avons utilisé des outils afin de faciliter la génération des données à partir des systèmes étudiés.

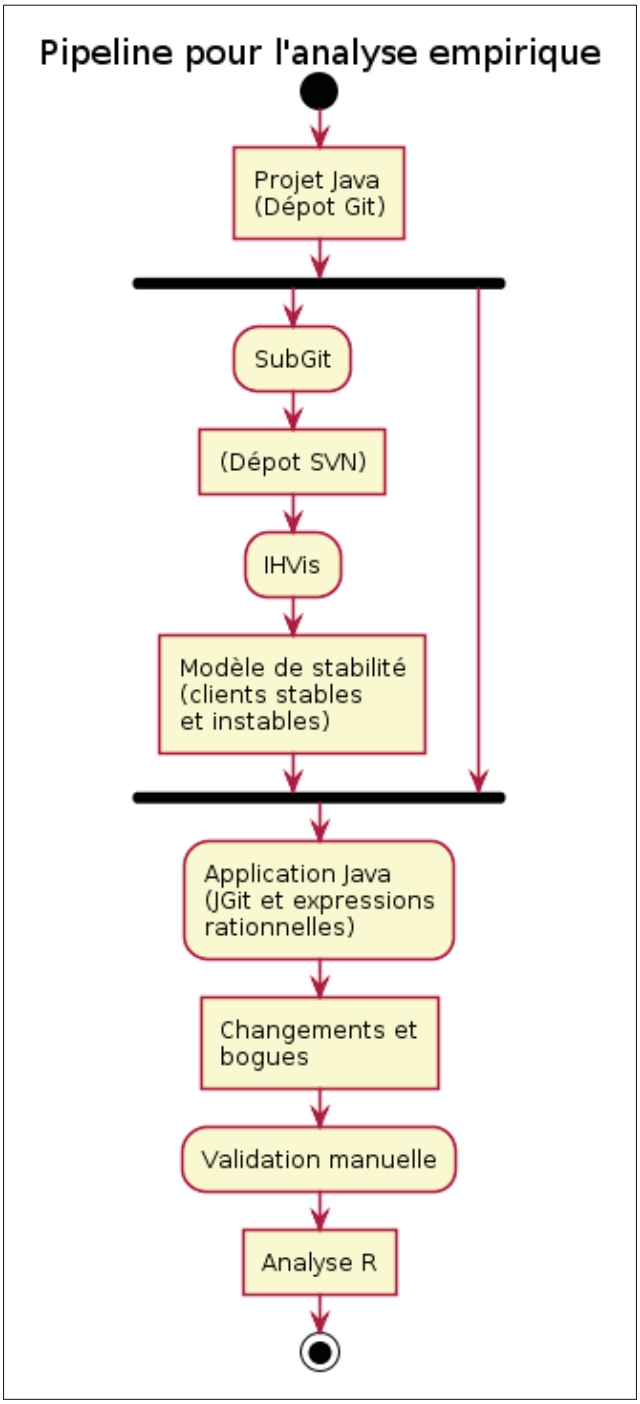

Figure 2.1 Processus de la collecte de données.

2.1.1 Outils utilisés

- IHvis est un outil de visualisation développé par Rufiange et Fuhrman (2014). Ce dernier permet de visualiser l'évolution des zones de variabilité et d'explorer les couplages stables et instables de chaque point de stabilité. L'outil inclut des fonctionnalités d'interaction telles que le zoom et le déplacement des éléments. La visualisation utilise aussi les notations UML pour représenter les éléments d'une zone de variabilité et les différents types de couplages entre eux. L'outil analyse les logiciels libres développés en Java et hébergés dans un système de gestion des versions utilisant SVN. Les données relatives à l'histoire de l'évolution entre les révisions sont extraites à partir du dépôt de code source. Ces données constituent les informations sur les changements effectués par les développeurs dans chaque révision. IHvis utilise un analyseur syntaxique afin de détecter tout type de changement dans plusieurs niveaux de granularité. Dans notre étude, nous avons utilisé l'outil IHvis pour générer la liste de tous les clients stables et instables dans une version logicielle.

- JGit [1] est une librairie implémentée en Java permettant de manipuler les systèmes de gestion des versions utilisant le logiciel Git. Le projet a été lancé en 2010 sous la licence publique de la fondation Eclipse. JGit offre plusieurs fonctionnalités telles que la lecture des informations sur les changements effectués dans un dépôt du code source, la création de nouvelles révisions ou de branches dans l'histoire des logiciels, etc. Nous avons utilisé cette librairie dans notre méthodologie afin de calculer le nombre de changements des clients stables et instables dans l'évolution des logiciels analysés.

- SubGit [2] est un outil de migration de dépôt SVN vers et à partir d'un dépôt Git. Cette solution bidirectionnelle donne aux développeurs la possibilité de changer le format du logiciel de gestion des versions de leurs projets. SubGit nous a permis de faire la migration des logiciels hébergés dans un dépôt Git. Cette contrainte vient de l'outil IHvis qui supporte seulement les dépôts du code source dans SVN.

1. https://eclipse.org/jgit/
2. http://www.subgit.com/

2.1.2 Génération du modèle de stabilité

La première étape dans le processus de la collecte de données consiste à générer la liste des clients stables et des clients instables en utilisant l'outil IHvis (Figure 2.1). La démarche consiste initialement à préparer le dépôt local en SVN du projet à étudier. Ensuite, pour chaque révision dans l'histoire du projet, nous avons identifié les clients des interfaces. À la fin du parcours de toutes les révisions, deux fichiers sont générés relatifs à la liste des clients stables et la liste des clients instables. Ces deux fichiers seront utilisés pour calculer le nombre de changements des clients et le nombre de leur implication dans des révisions de correction des bogues.

Dans le modèle de stabilité, il est nécessaire de spécifier des critères de classification des clients selon l'évolution et les modifications effectuées par les développeurs.

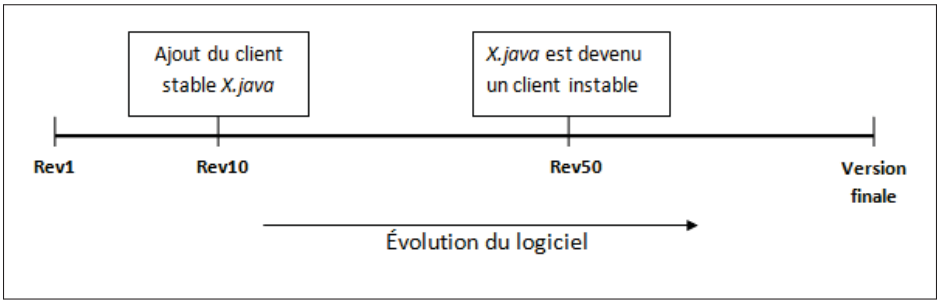

Figure 2.2 Changement du type des clients dans l'évolution des logiciels.

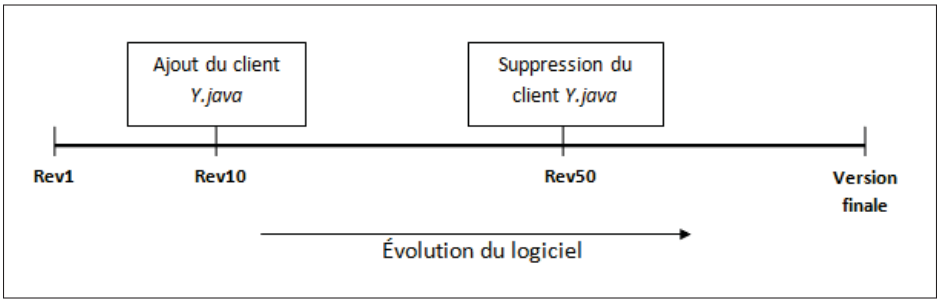

Figure 2.3 Suppression des clients dans l'évolution des logiciels.

Le premier cas est lié au changement du type des clients dans l'évolution des logiciels. Dans la Figure 2.2, la classe *X.java* a été ajoutée comme un client stable dans la révision 10. Ensuite, les développeurs ont introduit du couplage vers les implémentations dans la révision 50. Dans ce cas, cette classe joue le rôle d'un client stable entre les révisions 10 et 49, ainsi que le rôle d'un client instable entre la révision 50 et la version finale du logiciel.

Dans la Figure 2.3, les développeurs ont décidé de supprimer la classe *Y.java* du répertoire du code source. Toutefois, nous avons considéré seulement les classes qui existent dans la version finale. Dans ce cas, la classe *Y.java* ne figure pas dans la liste des clients générés.

2.1.3 Détection des changements au niveau des clients

La deuxième étape du processus (voir Figure 2.1) de la collecte des données consiste à calculer le nombre de changements des clients stables et des clients instables dans l'évolution des logiciels étudiés. Pour ce faire, nous avons utilisé la librairie JGit qui prend en entrée le dépôt local d'un projet sous format Git. Par la suite, nous avons mesuré pour chaque client le nombre des révisions dans lesquelles il a subi un changement. Cette mesure considère tout type de modification qui peut se produire dans une classe (par exemple, ajout de nouvelles fonctionnalités, renommage, correction de bogues). Les résultats sont générés dans des fichiers de type CSV. Nous avons aussi considéré dans les mesures deux cas de changement basés sur l'évolution des clients.

Dans l'exemple de la Figure 2.4, la classe *X.java* en tant que client stable a subi un changement dans la révision 15. Ensuite, cette dernière est devenue un client instable dans la révision 50, et elle a été modifiée dans la révision 100. Dans ce cas, le nombre de changements de la classe *X.java* en tant que client stable est égal à un, alors que le nombre de changements de cette dernière en tant que client instable est égal à deux.

Le deuxième cas de mesure concerne les modifications des noms des clients. Le renommage est une méthode de refactorisation appliquée dont le but est de clarifier les rôles des classes et des méthodes. Dans la Figure 2.5, le client *Y.java* est devenu *Z.java* dans la révision 25. Nous

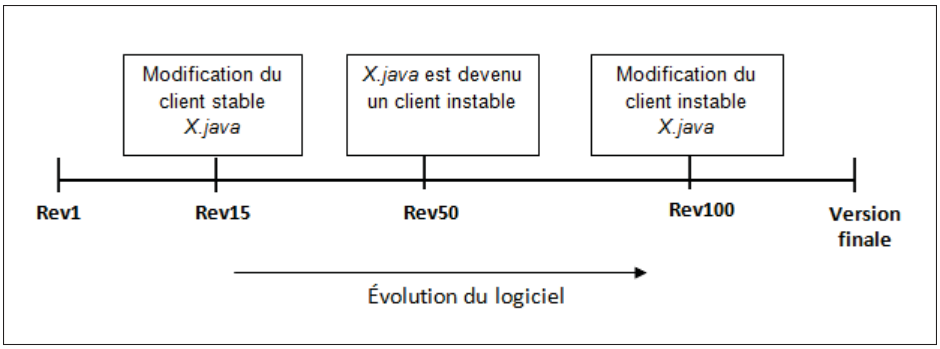

Figure 2.4 Modification du type des clients dans l'évolution des logiciels.

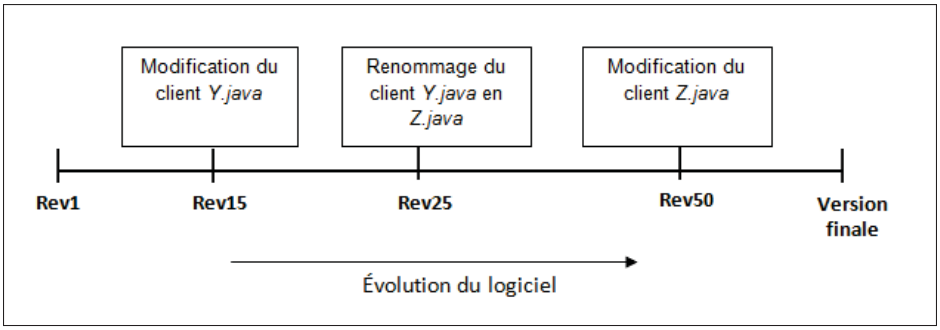

Figure 2.5 Renommage des clients dans l'évolution des logiciels.

avons considéré ce type de changement dans les mesures effectuées. Dans ce cas, la classe *Z.java* figure dans la liste des clients avec un total de changement égal à trois.

2.1.4 Détection des bogues au niveau des clients

Les bogues sont des défauts de conception ou des sources de dysfonctionnement pouvant affecter l'exécution normale d'un logiciel. Une révision de correction d'un bogue désigne les modifications apportées par les développeurs sur le code source afin de corriger les défauts identifiés. Les bogues sont souvent documentés dans des logiciels de suivi de problèmes tels que Bugzilla ou Jira. Nous avons utilisé des heuristiques et des expressions rationnelles. Ces méthodes nous ont permis d'identifier les révisions de corrections de bogues dans les systèmes étudiés afin de répondre à la deuxième question de recherche. Nous avons par la suite calculé

le nombre de changements liés aux corrections de bogues de chaque classe dans la liste des clients stables et instables.

Dans les systèmes de contrôle des versions, une révision contient un ensemble d'attributs tels que le message décrivant la tâche effectuée, la date de la création, les fichiers modifiés, le nom du développeur qui a créé la révision, etc. L'outil JGit nous a permis de parcourir la liste des révisions de chaque projet étudié et d'extraire les informations relatives à chaque révision. Nous avons analysé le message de chacune afin de savoir si les changements sont dans un contexte de correction de bogue. Les expressions régulières et les heuristiques implémentées dans notre méthode sont basées sur l'approche de Śliwerski *et al.* (2005). Cette dernière a été aussi utilisée dans plusieurs études sur la prédisposition des composants logiciels aux bogues (Jaafar *et al.* 2014b ; Khomh *et al.* 2012). L'analyse constitue premièrement à vérifier la correspondance entre le message et la liste des expressions rationnelles :

- **Expression 1** : (? = .*(fix | close | correct))

  La première expression est relative au verbe utilisé en anglais par les développeurs pour décrire les messages des révisions. Les trois verbes *fix, close, correct* analysés sont souvent utilisés pour indiquer que la révision constitue une correction d'un bogue. Lorsque le message d'une révision donnée contient une chaîne de caractère contenant l'un des verbes mentionnés, l'expression retourne vraie.

- **Expression 2** : (?=.*(bug | defect | error | patch | problem | issue))

  Les développeurs utilisent souvent une liste de mots spécifiques pour décrire une révision de correction de bogue. Dans le cas où le message d'une révision contient un de ces mots, la deuxième expression est donc validée.

- **Expression 3** : (?=.*[0-9]+)

  La troisième expression concerne l'existence d'une chaîne composée des caractères numériques. Les bogues sont souvent associés à un identifiant dans leur description dans les systèmes de suivi des problèmes. Pour faciliter la recherche des révisions liées à la correc-

tion des défauts, les développeurs ajoutent l'identifiant du bogue corrigé dans le message de la révision.

Nous avons aussi utilisé un score entre zéro et trois pour évaluer la correspondance des expressions rationnelles et les messages des révisions. Cette approche est basée sur les heuristiques de Śliwerski *et al.* (2005). Lorsque les trois expressions sont valides, le score trois est attribué au message. La révision est donc considérée comme une correction d'un bogue. Par exemple, le message "*Fixed bug 53784 : .class file missing from jar file export*" contient le verbe *Fixed*, le mot *bug* et la chaîne numérique 53784. Cette révision est automatiquement ajoutée à la liste des révisions de correction des bogues. Les révisions ayant un score égal à un ou deux sont ajoutées à l'ensemble des révisions qui nécessitent une vérification manuelle. Par exemple : le message "*Updated copyrights to 2016*" contient une chaîne numérique, mais il ne constitue pas une correction de défaut. Après la vérification manuelle, cette révision ne sera pas considérée comme un changement dans un contexte de correction de bogue. D'autre part, le message "*594961 : selecting a menu item did not execute*" contient seulement une chaîne numérique. Après une vérification dans le logiciel de suivi des problèmes, nous avons trouvé la description d'un bogue dont l'identifiant est 594961. Cette révision est donc considérée comme une correction d'un bogue. Parfois, les développeurs n'ajoutent pas les identifiants des bogues dans les messages des révisions (par exemple, *fix bug in layout*). Dans notre méthode d'identification des révisions, nous les avons considérés comme des révisions de correction de bogues.

Pour ArgoUML, les bogues sont documentés dans un système de suivi des problèmes [3]. Les développeurs de ce projet utilisent le mot défaut "*DEFECT*" afin d'indiquer que le type du problème est un bogue. Dans le processus d'identification des révisions de correction des défauts liés à ce système, nous avons vérifié chaque identifiant marqué dans les messages afin de s'assurer qu'il s'agit bien d'un changement lié à une correction de bogue. Les bogues dans JHotDraw sont aussi documentés dans un système de suivi des problèmes [4]. Ce dernier nous a permis de vérifier chaque identifiant détecté dans les messages des révisions de correction de

3. http://argouml.tigris.org/issues
4. https://sourceforge.net/p/jhotdraw/bugs/

bogue. Le logiciel Rhino possède aussi une base de données [5] des différents bogues documentés par les développeurs de ce projet. Ce système comporte un nombre considérable de bogues en raison de sa vaste utilisation.

2.2 Questions de recherche

Notre étude empirique vise à répondre à deux questions qui concernent la prédisposition des clients des interfaces ayant des couplages instables aux changements et aux bogues. Nous nous intéressons à analyser l'impact de l'introduction des relations de dépendance avec les implémentations sur la stabilité des clients dans l'évolution des logiciels.

RQ1 : Est-ce que le couplage instable affecte la prédisposition des clients aux changements dans l'évolution des logiciels ?

$H_{0_{RQ1}}$: Il n'y a aucune différence significative entre le couplage instable et le couplage stable en ce qui concerne la prédisposition des clients aux changements.

H_{a_1} : Les clients ayant du couplage instable sont plus exposés aux changements que les clients ayant du couplage stable.

La première question de notre étude empirique cherche à comprendre à quel point les couplages avec les implémentations affectent la prédisposition des clients aux changements dans l'évolution du logiciel. Pour y répondre, nous avons calculé dans l'histoire des systèmes le nombre de modifications des clients ayant seulement des couplages stables avec les interfaces et les clients ayant au moins un couplage instable avec une implémentation dans une zone de variabilité. Par la suite, nous avons comparé les résultats entre les deux types du couplage afin de déterminer s'il y a une différence significative. Cette comparaison nous permet également de rejeter l'hypothèse nulle $H_{0_{RQ1}}$ en faveur l'hypothèse alternative H_{a_1} et inversement.

RQ2 : Est-ce que le couplage instable affecte la prédisposition des clients aux bogues dans l'évolution des logiciels ?

5. https://bugzilla.mozilla.org/describecomponents.cgi?product=Rhino

$H_{0_{RQ2}}$: Il n'y a aucune différence significative entre le couplage instable et le couplage stable en ce qui concerne la prédisposition des clients aux bogues.

H_{a_2} : Les clients ayant du couplage instable sont plus exposés aux bogues que les clients ayant du couplage stable.

La deuxième question cherche à analyser l'impact du couplage instable sur la prédisposition des clients aux bogues dans l'évolution du logiciel. Nous avons calculé dans l'histoire des systèmes étudiés le nombre d'implications dans une correction de bogue des classes ayant seulement de couplages stables avec les interfaces et les classes ayant au moins un couplage instable avec une implémentation dans une zone de variabilité. Par la suite, nous avons comparé les résultats entre les deux types des clients afin de déterminer s'il y a une différence significative.

2.3 Variables de recherche

2.3.1 Variables indépendantes

Les deux variables indépendantes dans notre étude empirique sont les nombres de clients stables et de clients instables.

- Nombre de classes représentant des clients stables : l'ensemble des clients qui ont seulement des couplages avec les interfaces.
- Nombre de classes représentant des clients instables : l'ensemble des clients des interfaces qui ont simultanément des couplages avec les implémentations.

2.3.2 Variables dépendantes

Les deux variables dépendantes sont les mesures de la prédisposition des clients stables et instables aux changements et aux bogues :

- Prédisposition au changement : fait référence à tout type de changement qui se produit dans une classe. Les changements sont identifiés dans l'évolution des logiciels et calculés à partir des systèmes de contrôle des versions (SVN ou Git).
- Prédisposition au bogue : indique si une classe a subi au moins un changement dans un contexte de correction d'un bogue. Nous avons adopté une méthode d'identification des révisions de fixation de bogues basée sur des heuristiques et des expressions régulières largement utilisées dans la littérature.

2.4 Contexte de l'étude empirique

Nous avons examiné dans cette étude empirique trois systèmes qui varient selon le domaine d'application et la taille. Nous avons choisi d'étudier les logiciels JHotDraw, ArgoUML et Rhino. Ces systèmes ont été développés en Java, possèdent un code source libre et comprennent une longue période de développement.

De plus, les systèmes sélectionnés ont été étudiés par plusieurs chercheurs et dans différents domaines de recherche en génie logiciel. Nous soulignons aussi que cette étude est basée sur les versions majeures des projets mis en contexte. Par exemple, pour le système JHotDraw, nous avons étudié l'évolution des clients stables et instables dans les deux versions majeures JHotDraw 5 (5.1 - 5.4b2) et JHotDraw 7 (7.2 - 7.6). Ce choix dépend de l'importance du nombre de révisions et de la période du développement pour éviter l'inconsistance des résultats.

2.4.1 JHotDraw

JHotDraw [6] est un cadriciel de dessin graphique en 2D développé en Java. C'est un éditeur qui comporte plusieurs outils de dessin structuré. Il contient une palette d'outils pour dessiner des formes géométriques et insérer du texte. JHotDraw a été initialement développé par Erich Gamma qui est parmi les auteurs des patrons de conception GoF (Gamma *et al.*, 1994) et Thomas Eggenschwiler. Le projet a été lancé en octobre 2000 dont le but est de montrer l'appli-

6. http://www.jhotdraw.org/

cation des patrons de conception GoF dans un contexte réel. Le tableau 2.1 décrit les versions

Tableau 2.1 Liste des versions étudiées du projet JHotDraw.

systèmes	# révisions	# interfaces	# clients stables	# clients instables
JHotDraw 5	177	47	118	82
JHotDraw 7	349	63	231	74

majeures de JHotDraw étudiées dans notre recherche empirique. JHotDraw 5 est la première version majeure qui représente l'évolution du logiciel pendant cinq ans. Nous avons étudié 177 révisions entre les versions 5.2 et 5.4BETA2. Nous avons aussi identifié 47 interfaces, 118 clients stables et 82 clients instables.

JHotDraw 7 est la deuxième version majeure analysée dans notre étude. Nous n'avons pas considéré les versions entre 5.4B2 et 7.2 car les développeurs ont effectué une refactorisation de l'architecture du logiciel. Il nous a été difficile de générer le modèle de stabilité entre ces versions parce que l'outil IHvis ne détecte pas les opérations de renommage des classes et de restructuration des paquetages. JHotDraw 7 (entre 7.2 et 7.6) comporte 349 révisions, 63 interfaces, 231 clients stables et 74 clients instables.

Le projet JHotDraw a été largement étudié dans le domaine de la recherche en génie logiciel. Aversano *et al.* (2007) ont étudié la prédisposition aux changements des patrons de conception GoF dans l'évolution du JHotDraw 5. Di Penta *et al.* (2008) ont effectué la même étude de Aversano *et al.* (2007), mais en analysant la relation entre les rôles des classes dans les patrons et leur prédisposition aux changements. Khomh *et al.* (2009a) ont analysé dans JHotDraw 5.2B4 la relation entre le nombre de rôles des classes appartenant à un ou plusieurs patrons et leur prédisposition aux changements.

2.4.2 ArgoUML

ArgoUML [7] est un outil de modélisation UML supportant aussi la génération du code et l'ingénierie inverse. C'est un logiciel libre développé en Java et disponible en dix langues. Le projet a été lancé en 1999, et compte aujourd'hui plus de 19000 utilisateurs enregistrés et 150 développeurs.

Tableau 2.2 Liste des versions étudiées du projet ArgoUML.

systèmes	# révisions	# interfaces	# clients stables	# clients instables
ArgoUML 0.10	1088	43	91	12
ArgoUML 0.12	670	51	90	18
ArgoUML 0.14	1593	55	151	27
ArgoUML 0.16	1507	62	116	21
ArgoUML 0.18	1767	64	117	23
ArgoUML 0.20	1634	108	88	47
ArgoUML 0.22	1247	128	107	47
ArgoUML 0.24	899	131	108	47
ArgoUML 0.26	2850	128	108	46

ArgoUML comporte des versions stables et des versions de développement. Dans notre étude, nous avons choisi d'étudier les versions stables entre 0.10 et 0.26 (Tableau 2.2).

Dans la communauté de recherche en génie logiciel, ArgoUML a suscité l'intérêt de plusieurs chercheurs dans leurs études empiriques. Ce projet a été étudié par Aversano *et al.* (2007) et Di Penta *et al.* (2008) en analysant l'évolution des patrons GoF en ce qui concerne leur prédisposition aux changements. Khomh *et al.* (2012) ont analysé la relation entre les classes qui implémentent des anti-patrons et leur prédisposition aux changements et aux bogues.

7. http://argouml.tigris.org/

2.4.3 Rhino

Rhino [8] est un moteur d'interprétation du langage JavaScript géré par la fondation Mozilla. Il permet notamment de convertir les scripts en des classes Java. Rhino a été intégré par défaut dans la version standard de Java SE6.

Tableau 2.3 Liste des versions étudiées du projet Rhino.

systèmes	# révisions	# interfaces	# clients stables	# clients instables
Rhino 1.5R1	369	34	34	33
Rhino 1.5R2	262	41	41	39
Rhino 1.5R3	86	42	47	41
Rhino 1.5R4.1	362	44	50	44
Rhino 1.5R5	364	45	50	41
Rhino 1.6R1	360	49	56	50
Rhino 1.6R4	160	51	56	52
Rhino 1.6R6	174	52	63	53
Rhino 1.7R3	580	63	98	73
Rhino 1.7R4	239	63	98	73

Le tableau 2.3 décrit les versions étudiées dans notre recherche empirique. Nous avons analysé l'évolution de ce projet entre la version 1.5R1 (Mai 1999) et la version 1.7R4 (Juin 2012). En raison de sa vaste utilisation, Rhino possède un nombre considérable de bogues qui sont documentés dans son système de suivi des problèmes. Les chercheurs intéressés par l'étude des défauts de conception ont souvent incorporé ce projet dans leurs approches. Eaddy *et al.* (2008) ont analysé le lien entre les préoccupations transversales et l'introduction de bogues dans Rhino. Jaafar *et al.* (2014b) ont étudié la mutation des anti-patrons dans les classes logicielles et leur prédisposition aux bogues. Khomh *et al.* (2012) ont réalisé une étude empirique sur la prédisposition des classes avec des anti-patrons aux changements et aux bogues.

8. http://www.mozilla.org/Rhino

2.5 Méthodes d'analyse

Les approches empiriques sont des méthodes de recherche basées sur l'exploration des données issues de l'observation et de l'expérimentation. Ce type d'étude s'appuie principalement sur le test de la validité des hypothèses pour former une conclusion sur la population étudiée. Or, les chercheurs utilisent souvent des tests statistiques tels que les tests paramétriques ou les tests non paramétriques comme un moyen de vérification pouvant soutenir ou rejeter une hypothèse de recherche.

2.5.1 Test de Mann–Whitney

Le test de Mann-Witney (Mann et Whitney, 1947) est un test statistique non paramétrique qui permet de comparer deux échantillons de données indépendants. La comparaison des deux distributions permet également d'accepter l'hypothèse alternative lorsqu'il y a une différence significative, ou de la rejeter en faveur de l'hypothèse nulle lorsque les deux échantillons sont identiques en terme de position. Les tests non paramétriques sont généralement utilisés lorsqu'il n'y a pas de condition sur les formes de données, et aussi dans le cas de la non-normalité des variables d'études ou d'échantillons de faible taille. Le test de Mann-Whitney est applicable sur les échantillons de données non appariées où les observations des deux distributions sont totalement indépendantes. À la différence des tests paramétriques, le calcul dans ce test porte sur les rangs attribués suite au classement des valeurs par ordre croissant. Les avantages des tests non paramétriques reposent sur leur facilité d'utilisation, leur application sur des échantillons de taille réduite, ainsi que leur utilité lorsque les applications des autres méthodes statistiques ne sont pas satisfaites.

Nous avons appliqué dans notre méthodologie de recherche le test de Mann-Whitney sur les données issues de différents projets étudiés afin de répondre à nos hypothèses. Les deux populations étudiées sont le nombre de changements des clients stables et le nombre de changements des clients instables. Le choix de cette technique repose principalement sur l'existence des versions dont les groupes de données possèdent une faible taille, l'indépendance entre les

observations des deux échantillons, ainsi que les distributions des données ne suivent pas la loi normale.

2.5.2 Taille d'effet

En statistique, la taille d'effet est une mesure quantitative de la force de relation entre les variables. Elle est souvent complémentaire à d'autres mesures statistiques. La taille d'effet est considérée comme la distance entre l'hypothèse nulle et l'hypothèse alternative. Plus sa valeur est importante, plus l'écart à l'hypothèse nulle est large.

Le delta de Cliff (d) (Cliff, 1993) est une mesure non paramétrique de la taille d'effet proposé par Norman Cliff. Cette dernière permet de mesurer la différence entre les valeurs des deux distributions de données. Les valeurs possibles de delta de Cliff sont dans l'intervalle [-1,1]. La différence est considérée négligeable si $d < 0.148$, faible si $0.148 \leq d < 0.33$, moyenne si $0.33 \leq d < 0.474$, et large si $d \geq 0.474$. Nous avons appliqué la mesure de delta de Cliff sur les données obtenues lorsque le test de Mann-Whitney donne une valeur significative pour chaque version des systèmes étudiés.

2.5.3 Diagramme en boîte à moustaches

Le diagramme en boîte à moustaches (McGill *et al.*, 1978) est une représentation graphique des données statistiques permettant de visualiser plusieurs paramètres de la distribution d'une variable donnée tels que la médiane, les quartiles, les valeurs minimales et maximales, etc. Ce diagramme est généralement utilisé pour décrire d'une façon sommaire un échantillon d'étude. Toutefois, il ne fournit pas assez de détails tels que la forme de la distribution, mais ils peuvent être utiles pour comparer deux populations de taille différente.

La Figure 2.6 illustre un exemple des boîtes à moustaches de deux échantillons de données. La partie centrale de la boîte couvre la moitié centrale des données et va du premier quartile Q1 (25% des effectifs) au troisième quartile Q3 (75% des effectifs). La ligne horizontale dans la boîte représente la médiane Q2 (50% des effectifs). Les deux moustaches inférieures et

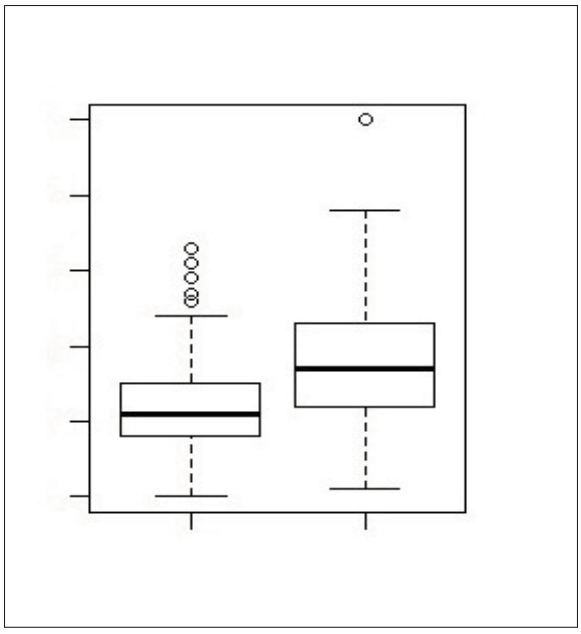

Figure 2.6 Exemple de représentation des distributions de données avec des boîtes à moustaches.

supérieures permettent de délimiter les valeurs adjacentes qui sont déterminées à partir de l'écart interquartile (Q3 - Q1). Les données aberrantes situées au-delà des valeurs adjacentes sont représentées par des cercles. Nous avons utilisé les boîtes à moustaches pour représenter et comparer les distributions des données relatives au nombre de changements des clients des interfaces.

CHAPITRE 3

RÉSULTATS ET DISCUSSIONS

Dans ce chapitre, nous présentons les résultats obtenus dans notre étude empirique. Ces derniers concernent l'application des tests statistiques sur les données générées à partir des projets étudiés. Dans la première section, nous présentons une analyse statistique des données collectées. Ensuite, nous discutons les résultats pour chaque système. Enfin, nous présentons les obstacles à la validité et les limites de notre approche.

3.1 RQ1 : Est-ce que le couplage instable affecte la prédisposition des clients aux changements dans l'évolution des logiciels ?

La première question de l'étude empirique concerne le lien entre le couplage avec les artefacts instables protégés par les interfaces et la prédisposition des clients aux changements. Pour répondre à cette dernière, nous avons suivi un processus de collecte et d'analyse de données obtenues à partir de différents projets étudiés. Les tests statistiques utilisés dans la méthodologie de recherche ont été appliqués sur les distributions des nombres de changements des clients stables et des clients instables. Le test non paramétrique de Mann-Whitney nous a permis en premier lieu de comparer les deux distributions de données afin de rejeter l'hypothèse alternative H_{a_1} ou l'hypothèse nulle $H_{0_{RQ1}}$ lorsqu'il y a une différence significative. Nous avons fixé le seuil de la valeur-p à 5% (p-value ≤ 0.05) pour accepter l'hypothèse alternative. Nous avons aussi utilisé le delta de Cliff comme un test complémentaire pour mesurer la taille d'effet. Lorsque la valeur-p donne une valeur significative, la différence peut être faible, moyenne ou large.

La présente section décrit les résultats statistiques obtenus à partir des données recueillies sur les trois systèmes étudiés, et dont l'objectif est de répondre à la première question de l'étude empirique.

3.1.1 JHotDraw

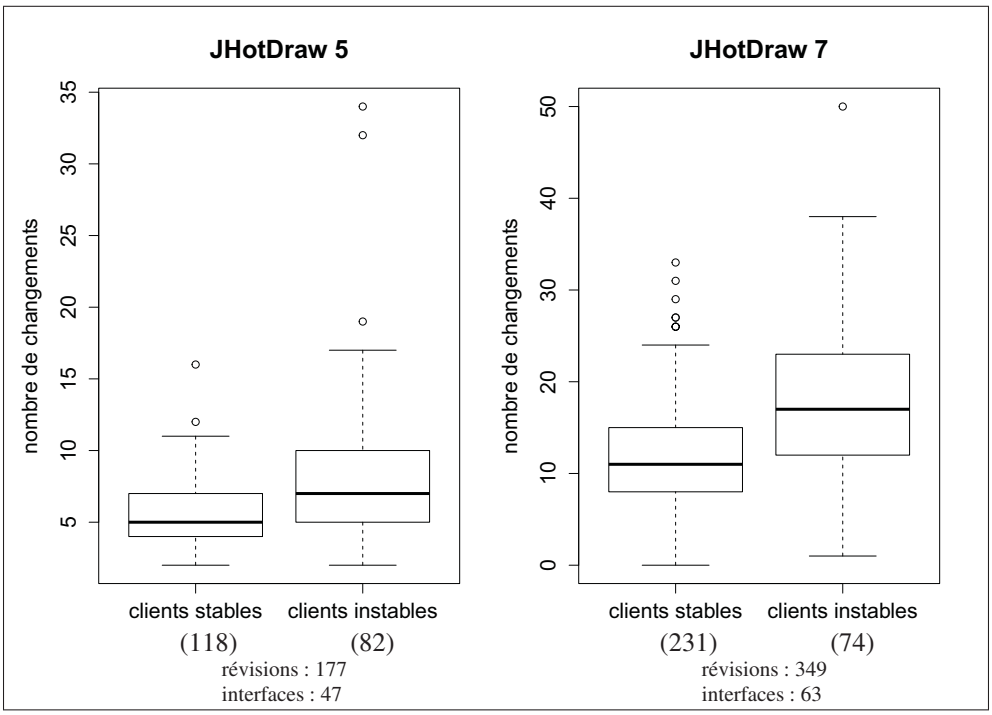

Figure 3.1 Boîtes à moustaches des nombres de changements des clients stables et instables dans JHotDraw.

La Figure 3.1 présente les boîtes à moustaches relatives aux données empiriques des changements des clients dans l'évolution du JHotDraw. Cette représentation a été utilisée pour comparer les deux distributions de données dans les versions étudiées. Nous observons que dans les deux versions de JHotDraw, les clients instables ont des pourcentages de changements plus élevés par rapport aux clients stables.

Tableau 3.1 Résultats statistiques de la prédisposition des clients (instables vs stables) aux changements dans JHotDraw.

versions	**valeur-p**	**delta de Cliff**
JHotDraw 5	<0.0001	faible (0.30)
JHotDraw 7	<0.0001	large (0.52)

Le tableau 3.1 décrit les résultats de l'application du test de Mann-Whitney sur les données du système JHotDraw. L'analyse statistique indique qu'il y a une différence significative dans les deux versions étudiées. Dans JHotDraw 5, la différence entre le nombre de changements des clients stables et les clients instables est statistiquement significative (valeur-p = $1.9{\cdot}10^{-4}$), alors que la mesure de delta de Cliff est faible. Dans JHotDraw 7, l'analyse a donné une différence largement significative (valeur-p = $1.06{\cdot}10^{-11}$). De plus, la valeur de delta de Cliff a donné une taille d'effet large. D'où l'écart à l'hypothèse nulle est plus grand.
D'après les résultats statistiques obtenus dans JHotDraw, nous pouvons alors rejeter la première hypothèse nulle $H_{0_{RQ1}}$, et nous concluons qu'il y a une relation entre le couplage avec les implémentations des interfaces et la prédisposition des classes clients aux changements.

3.1.2 ArgoUML

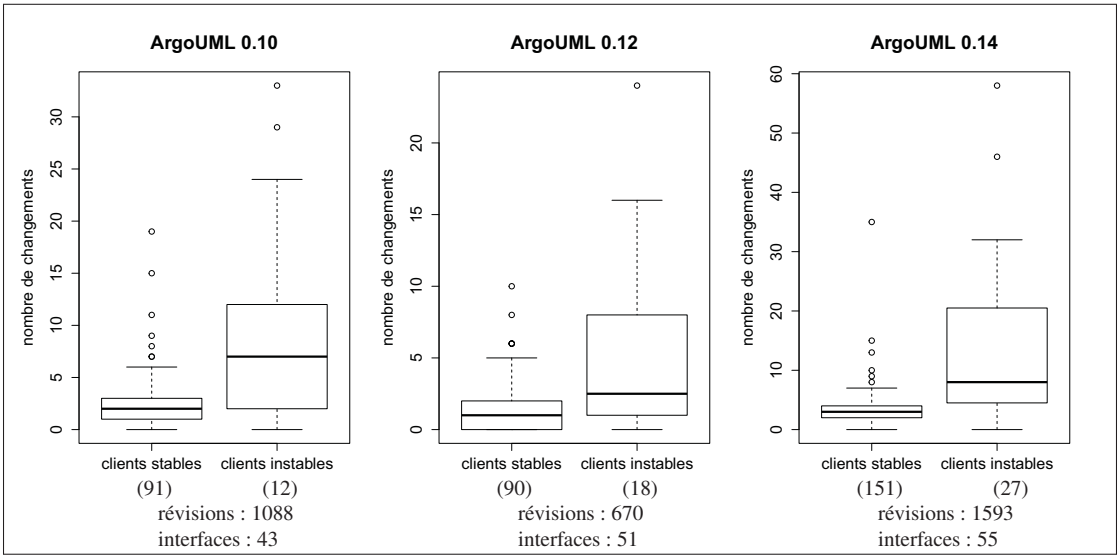

Figure 3.2 Boîtes à moustaches des nombres de changements des clients stables et instables dans ArgoUML (0.10 - 0.14).

Les Figures 3.2, 3.3 et 3.4 illustrent les boîtes à moustaches permettant de comparer les distributions des nombres de changements des clients stables et des clients instables de toutes les versions étudiées. Nous constatons la présence d'une différence considérable entre les deux

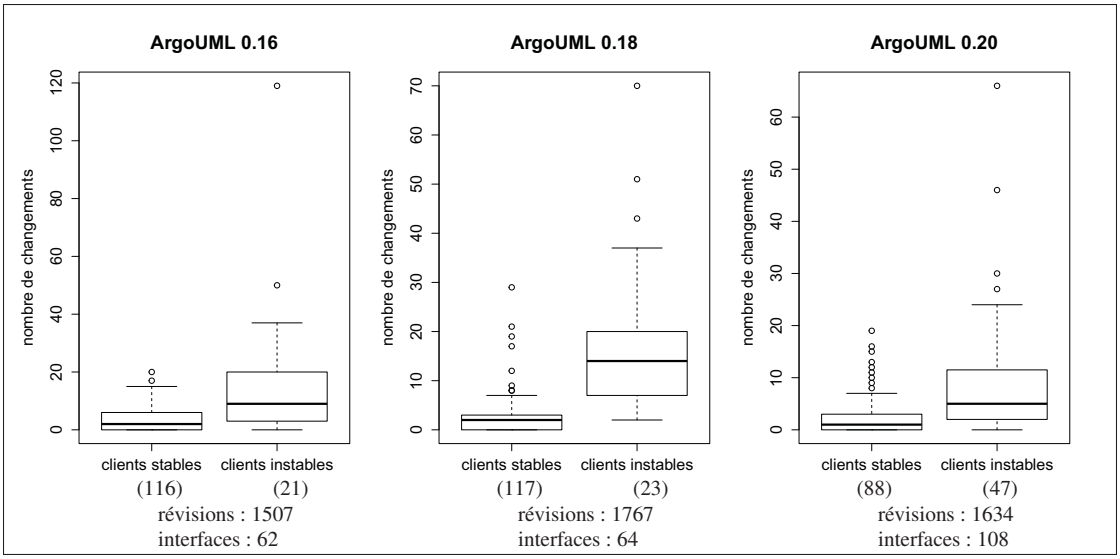

Figure 3.3 Boîtes à moustaches des nombres de changements des clients stables et instables dans ArgoUML (0.16 - 0.20).

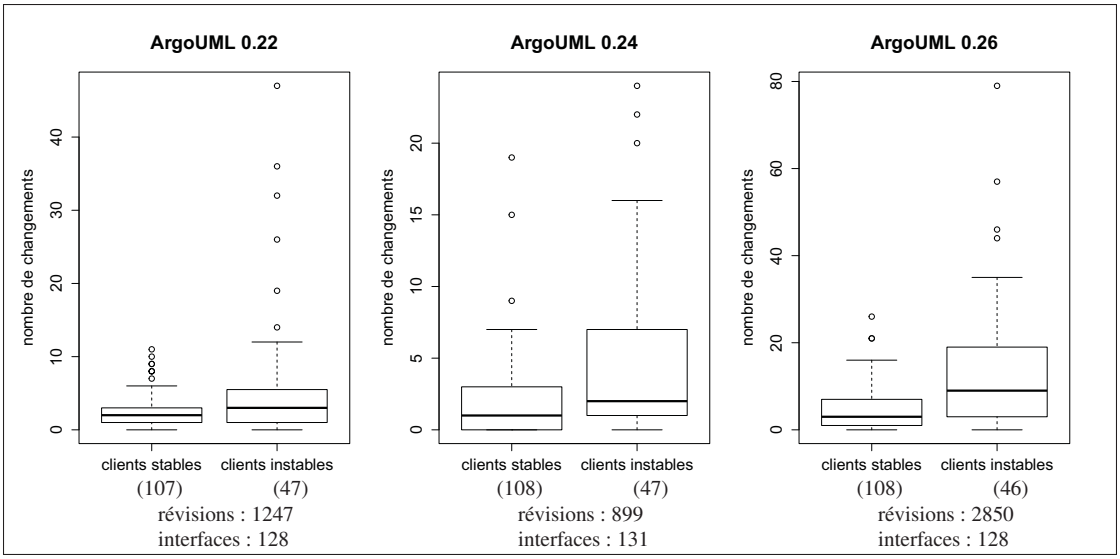

Figure 3.4 Boîtes à moustaches des nombres de changements des clients stables et instables dans ArgoUML (0.22 - 0.26).

échantillons analysés. Dans les versions 0.10, 0.12, 0.14, 0.18, et 0.26 de ArgoUML, les clients ayant des couplages avec les variations des interfaces ont des valeurs de changement plus élevées que celles des clients ayant seulement des dépendances stables. Dans le reste des versions,

les clients instables ont une fréquence de changement légèrement plus élevé que les clients stables.

Le tableau 3.2 résume les résultats statistiques de chaque version étudiée dans le projet ArgoUML. Nous constatons que dans toutes les versions (0.10 - 0.20), la différence est statistiquement significative (valeur-p < 0.05). Les clients instables sont plus exposés aux changements par rapport aux clients stables. Les résultats des versions 0.24 et 0.26 sont aussi significatifs avec une taille d'effet moyenne. Toutefois, la version 0.22 possède une faible valeur de delta de Cliff. Dans le reste des versions, la taille d'effet est large.

En fonction des résultats statistiques obtenus dans ArgoUML, nous concluons que l'hypothèse alternative H_{a_1} est acceptée et nous rejetons l'hypothèse nulle $H_{0_{RQ1}}$ de la première question de recherche.

Tableau 3.2 Résultats statistiques de la prédisposition des clients (instables vs stables) aux changements dans ArgoUML.

versions	**valeur-p**	**delta de Cliff**
ArgoUML 0.10	0.002	large (0.50)
ArgoUML 0.12	<0.0001	large (0.48)
ArgoUML 0.14	<0.0001	large (0.66)
ArgoUML 0.16	<0.0001	large (0.50)
ArgoUML 0.18	<0.0001	large (0.86)
ArgoUML 0.20	<0.0001	large (0.51)
ArgoUML 0.22	0.04	faible (0.19)
ArgoUML 0.24	<0.0001	moyen (0.36)
ArgoUML 0.26	<0.0001	moyen (0.39)

3.1.3 Rhino

Les Figures 3.5, 3.6 et 3.7 présentent les boîtes à moustaches concernant les distributions des nombres de changements des clients instables et des clients stables du projet Rhino. Nous

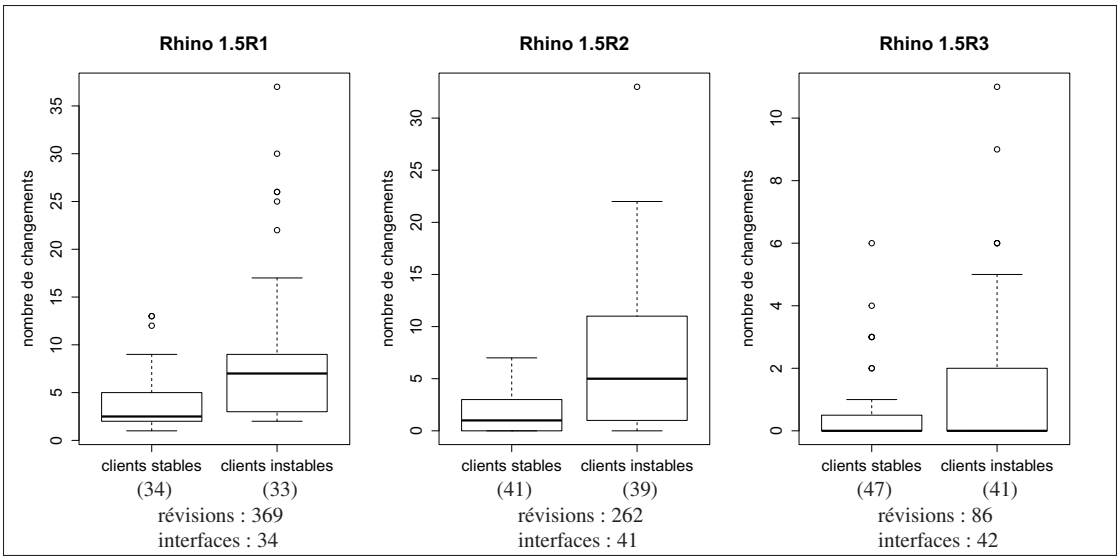

Figure 3.5 Boîtes à moustaches des nombres de changements des clients stables et instables dans Rhino (1.5R1 - 1.5R3).

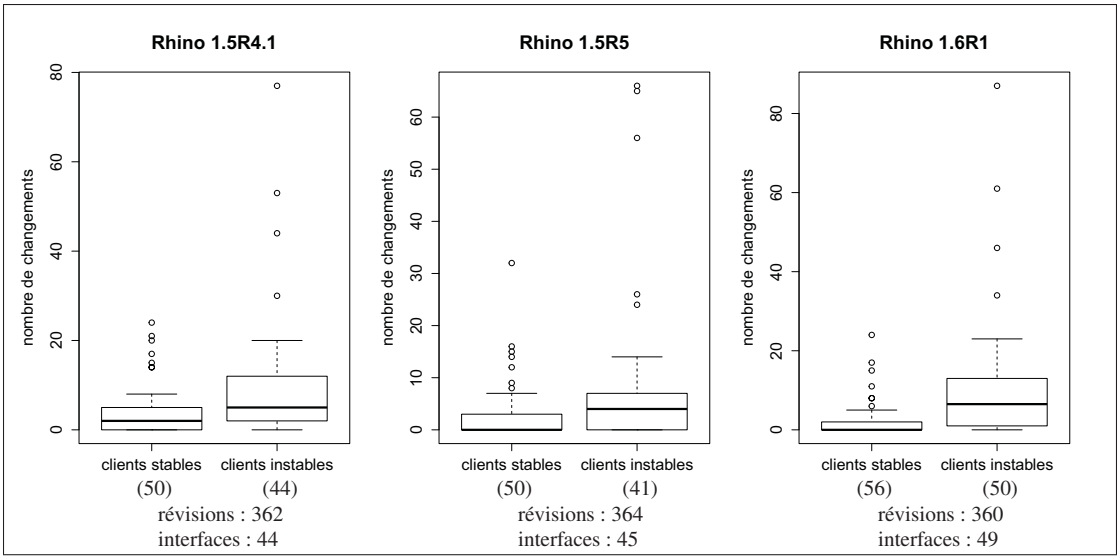

Figure 3.6 Boîtes à moustaches des nombres de changements des clients stables et instables dans Rhino (1.5R4.1 - 1.6R1).

constatons que dans toutes les versions étudiées, le pourcentage de changements des clients instables est supérieur par rapport au pourcentage de changement des clients stables.

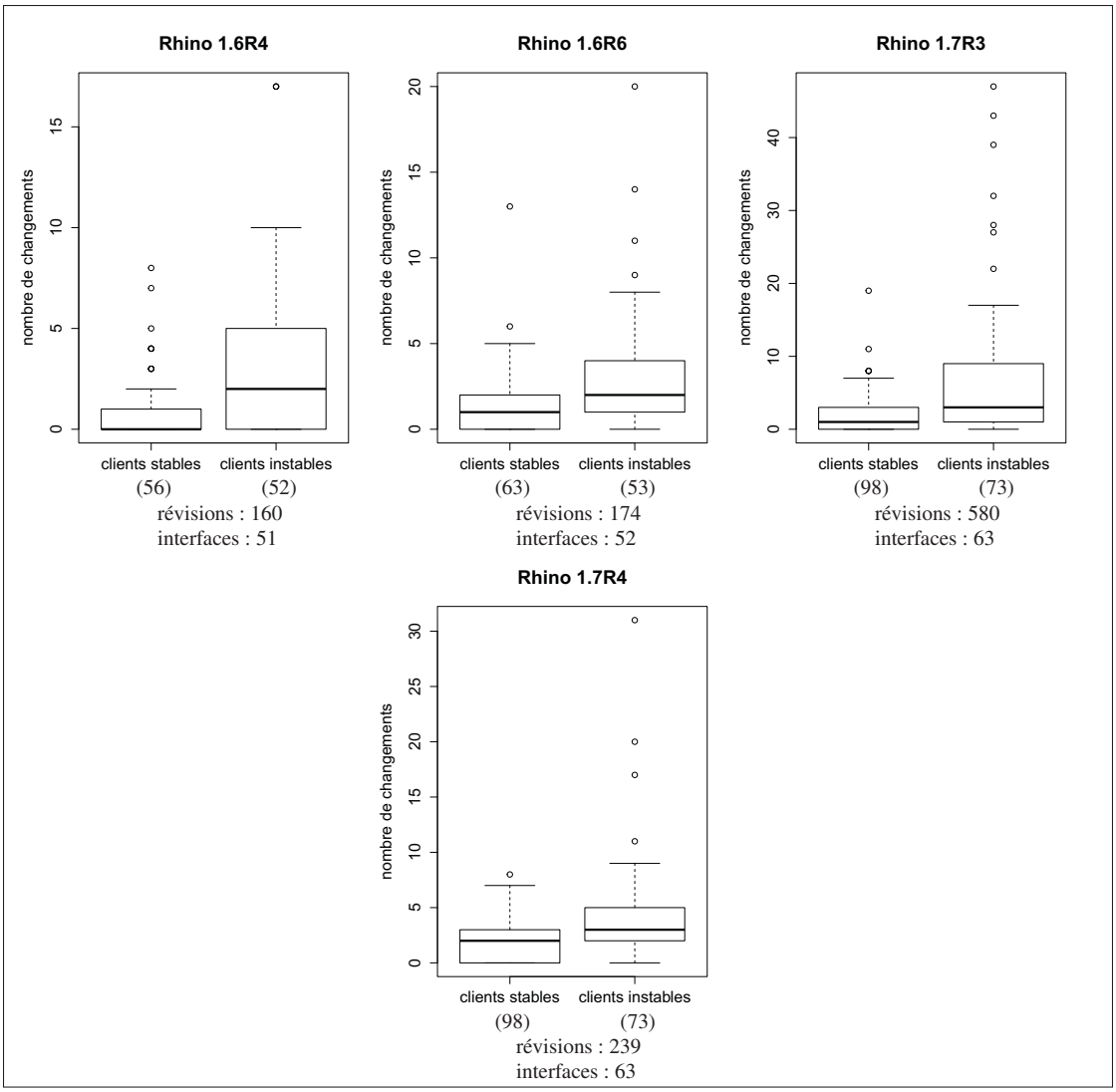

Figure 3.7 Boîtes à moustaches des nombres de changements des clients stables et instables dans Rhino (1.6R4 - 1.7R4).

Nous présentons dans le tableau 3.3 les résultats statistiques des données empiriques du système Rhino. Une première observation concerne la valeur-p dans les versions étudiées. Cette dernière est statistiquement significative dans toutes les versions. Dans l'évolution du logiciel Rhino, les clients instables ont subi plus de changements que les clients stables. De plus, la mesure de delta de Cliff a donné un effet qui varie entre moyen et faible.

L'hypothèse nulle $H_{0_{RQ1}}$ de la première question est donc rejetée en faveur de l'hypothèse alternative H_{a_1}. Nous pouvons conclure que la différence entre les valeurs des changements

Tableau 3.3 Résultats statistiques de la prédisposition des clients (instables vs stables) aux changements dans Rhino.

versions	**valeur-p**	**delta de Cliff**
Rhino 1.5R1	<0.0001	moyen (0.43)
Rhino 1.5R2	<0.0001	moyen (0.45)
Rhino 1.5R3	0.04	faible (0.20)
Rhino 1.5R4.1	0.01	faible (0.29)
Rhino 1.5R5	0.006	faible (0.31)
Rhino 1.6R1	<0.0001	moyen (0.46)
Rhino 1.6R4	<0.0001	moyen (0.40)
Rhino 1.6R6	<0.0001	faible (0.31)
Rhino 1.7R3	<0.0001	moyen (0.35)
Rhino 1.7R4	<0.0001	faible (0.29)

est significative. Les dépendances des implémentations rendent les clients des interfaces plus exposés aux changements par rapport aux clients ayant seulement du couplage stable.

3.2 RQ2 : Est-ce que le couplage instable affecte la prédisposition des clients aux bogues dans l'évolution des logiciels

Rappelons que la deuxième question de l'étude empirique analyse la relation entre le couplage vers les implémentations et la prédisposition des clients aux bogues. Pour y répondre, nous avons suivi un processus de collecte de données liées aux bogues basé sur des méthodes largement utilisées dans la littérature. Pour chaque système étudié, nous avons identifié les révisions de correction des défauts signalés par les développeurs dans l'évolution logicielle. Nous avons par la suite détecté les classes clients qui ont été modifiées dans ces révisions. Nous avons aussi appliqué le test de Mann-Whitney pour l'analyse statistique des données obtenues.

3.2.1 JHotDraw

La Figure 3.8 présente les pourcentages de correction de bogues dans l'ensemble des clients stables et des clients instables. Nous observons que dans la version JHotDraw 5, il n'existe pas de différence entre les deux distributions de données. Tandis que dans la version JHotDraw 7,

les clients instables ont des valeurs de bogues légèrement supérieures par rapport aux clients stables.

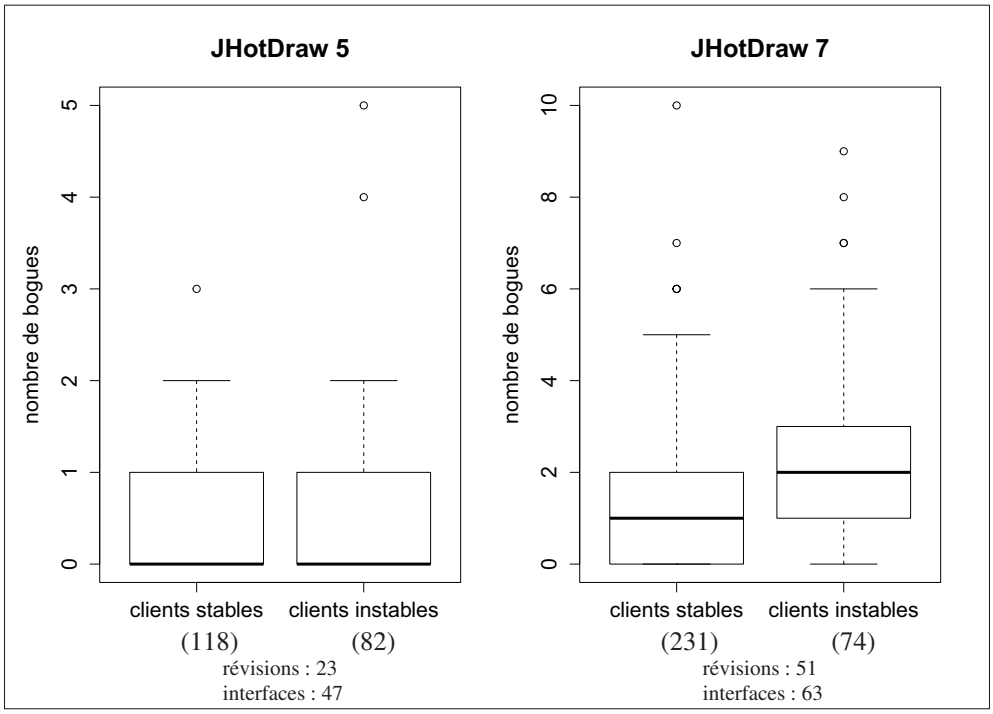

Figure 3.8 Boîtes à moustaches des nombres de corrections de bogues des clients stables et instables dans JHotDraw.

Tableau 3.4 Résultats statistiques de la prédisposition des clients (instables vs stables) aux bogues dans JHotDraw.

versions	**valeur-p**	**delta de Cliff**
JHotDraw 5	0.06	négligeable (0.18)
JHotDraw 7	<0.0001	moyen (0.33)

Le tableau 3.4 présente les résultats statistiques de l'application du test de Mann-Whitney. Nous constatons que la valeur-p est significative dans la version JHotDraw 7 avec une taille d'effet moyenne. Cependant, le test n'a pas donné de résultat significatif dans la version JHotDraw 5.

Selon ces résultats, nous concluons qu'il existe une relation entre le couplage avec les implémentations et la prédisposition des clients aux bogues dans seulement la version JHotDraw 7. Nous pouvons donc rejeter l'hypothèse nulle $H_{0_{RQ2}}$ en faveur de l'hypothèse alternative H_{a_2}.

3.2.2 ArgoUML

Les Figures 3.9, 3.10 et 3.11 montrent qu'il existe une différence significative entre les valeurs de changement des clients stables et des clients instables dans presque toutes les versions étudiées du système ArgoUML. Dans les versions 0.12, 0.14 et 0.18, les clients instables ont des pourcentages de bogues plus élevés par rapport aux clients stables. Dans le reste des versions analysées, la différence entre les deux types des clients est légèrement considérable.

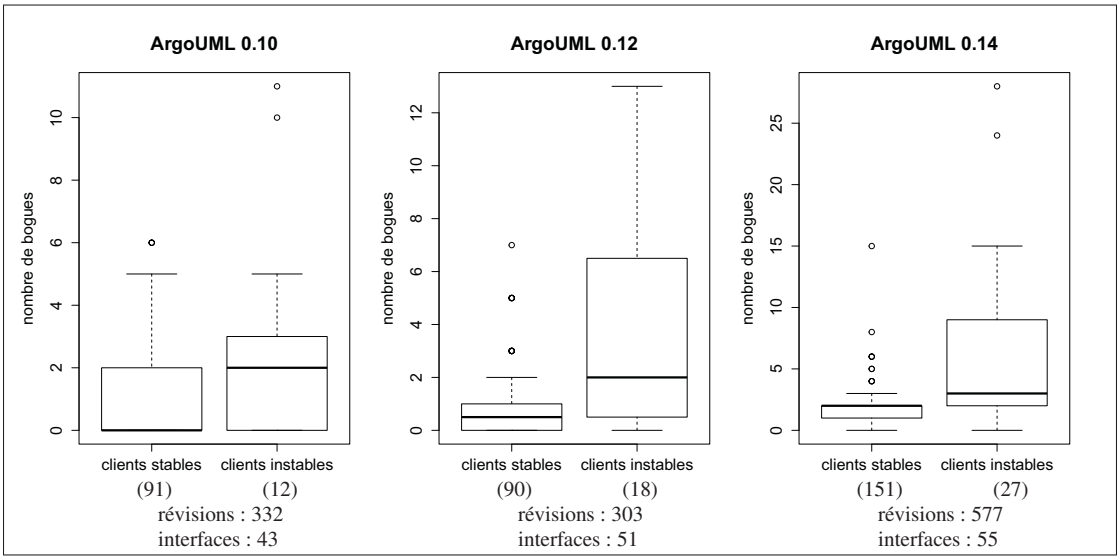

Figure 3.9 Boîtes à moustaches des nombres de corrections de bogues des clients stables et instables dans ArgoUML (0.10 - 0.14).

Le tableau 3.5 décrit les résultats statistiques obtenus dans le projet ArgoUml. Nous observons que la valeur-p est significative dans toutes les versions analysées, ainsi que l'effet varie entre moyen et large pour la majorité de versions. Dans l'évolution de ArgoUml, les clients instables ont subi plus de changements dans un contexte de correction de bogues par rapport aux clients stables. Dans le cas du projet ArgoUML, l'hypothèse alternative H_{a_2} est acceptée

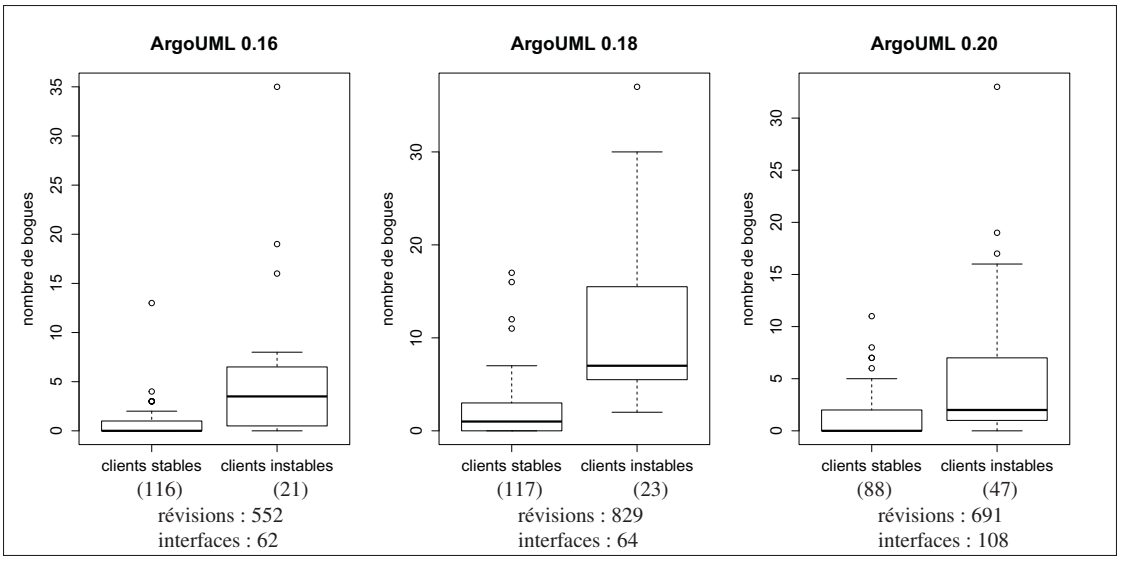

Figure 3.10 Boîtes à moustaches des nombres de corrections de bogues des clients stables et instables dans ArgoUML (0.16 - 0.20).

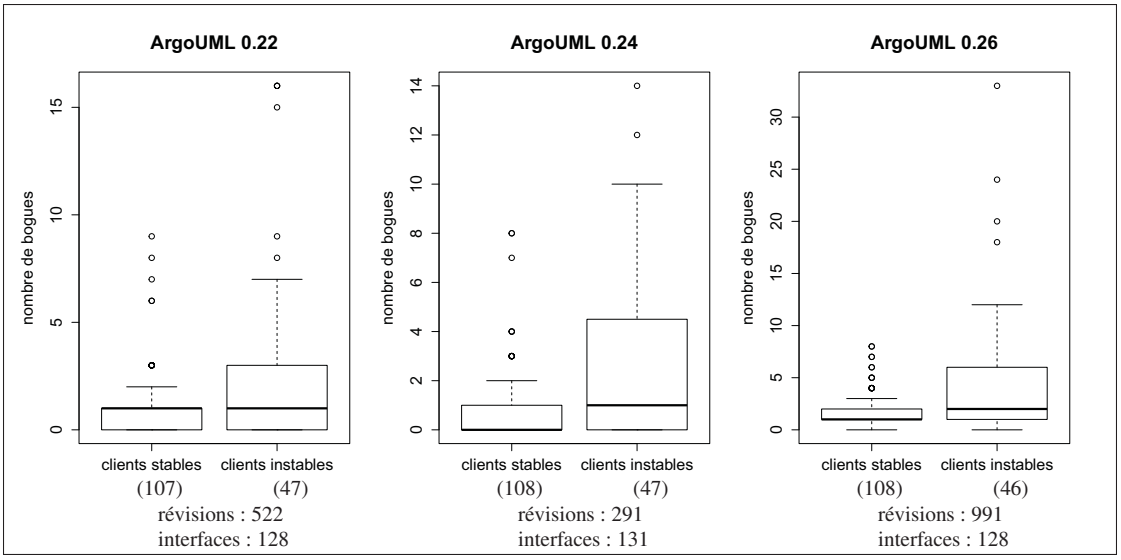

Figure 3.11 Boîtes à moustaches des nombres de corrections de bogues des clients stables et instables dans ArgoUML (0.22 - 0.26).

et nous rejetons l'hypothèse nulle $H_{0_{RQ2}}$. Nous pouvons donc valider que le couplage avec les implémentations ait un impact sur l'implication des clients dans les bogues.

Tableau 3.5 Résultats statistiques de la prédisposition des clients (instables vs stables) aux bogues dans ArgoUML.

versions	valeur-p	delta de Cliff
ArgoUML 0.10	0.01	moyen (0.35)
ArgoUML 0.12	0.002	moyen (0.42)
ArgoUML 0.14	<0.0001	large (0.49)
ArgoUML 0.16	<0.0001	large (0.52)
ArgoUML 0.18	<0.0001	large (0.83)
ArgoUML 0.20	<0.0001	large (0.52)
ArgoUML 0.22	0.02	faible (0.21)
ArgoUML 0.24	<0.0001	moyen (0.35)
ArgoUML 0.26	<0.0001	moyen (0.37)

3.2.3 Rhino

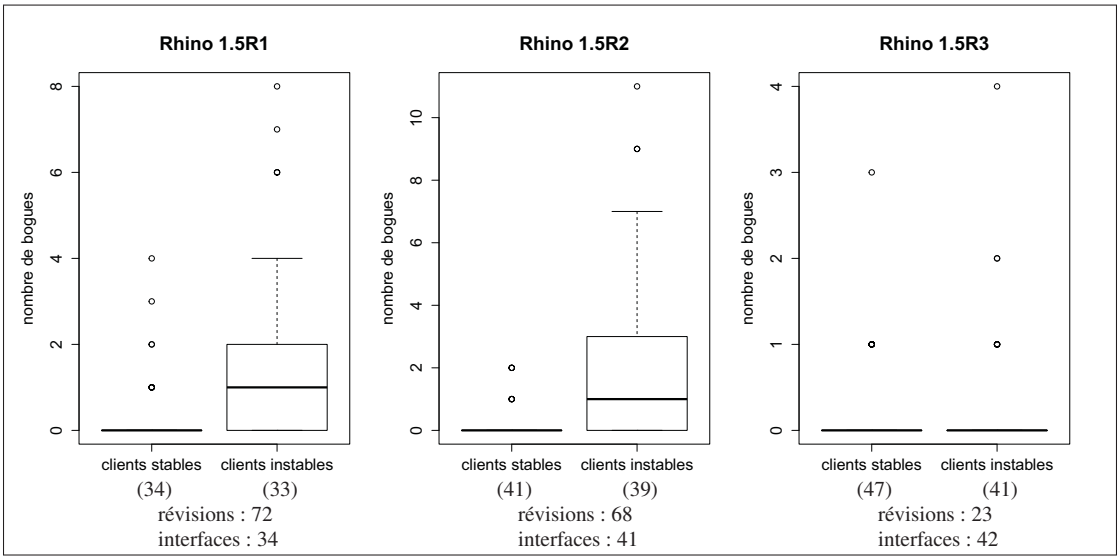

Figure 3.12 Boîtes à moustaches des nombres de corrections de bogues des clients stables et instables dans Rhino (1.5R1 - 1.5R3).

Les Figures 3.12, 3.13 et 3.14 présentent les nombres de bogues des clients instables et des clients stables du projet Rhino. Nous constatons que les clients instables ont des nombres de corrections de bogues légèrement supérieurs par rapport aux clients stables dans la majorité des versions étudiées. Toutefois, la différence est faible dans les versions 1.5R3 et 1.7R4.

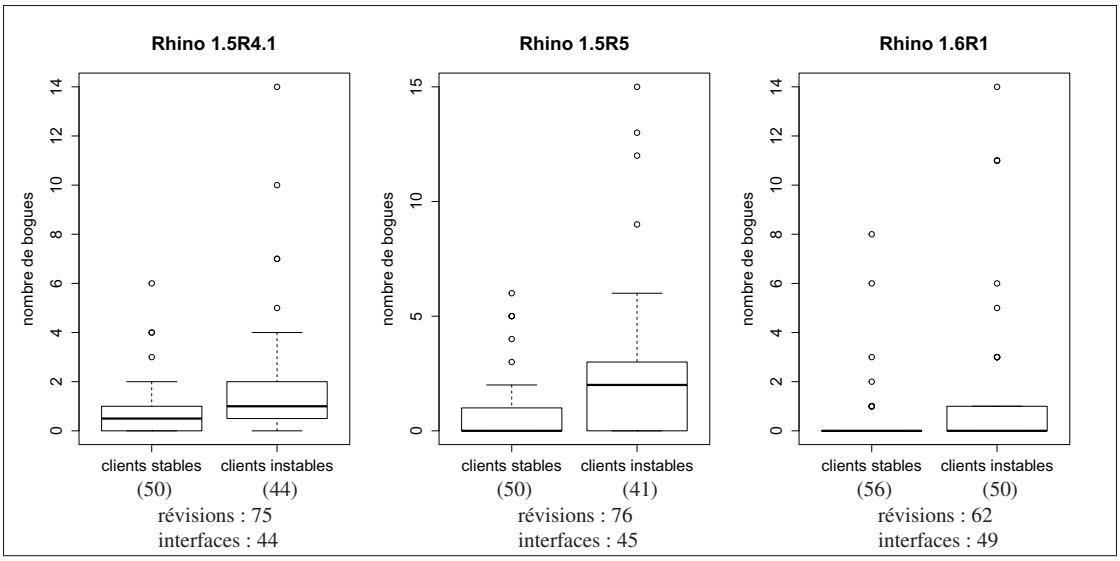

Figure 3.13 Boîtes à moustaches des nombres de corrections de bogues des clients stables et instables dans Rhino (1.5R4.1 - 1.6R1).

Tableau 3.6 Résultats statistiques de la prédisposition des clients aux bogues dans Rhino.

versions	**valeur-p**	**delta de Cliff**
Rhino 1.5R1	<0.0001	moyen (0.41)
Rhino 1.5R2	<0.0001	moyen (0.45)
Rhino 1.5R3	0.3	négligeable (0.07)
Rhino 1.5R4.1	<0.0001	faible (0.31)
Rhino 1.5R5	<0.0001	moyen (0.38)
Rhino 1.6R1	<0.0001	faible (0.23)
Rhino 1.6R4	<0.0001	moyen (0.36)
Rhino 1.6R6	<0.0001	faible (0.32)
Rhino 1.7R3	<0.0001	faible (0.24)
Rhino 1.7R4	<0.0001	faible (0.23)

Le tableau 3.6 présente les résultats obtenus à partir de l'application des tests statistiques sur les données issues du projet Rhino. Nous constatons que la valeur-p est non significative dans la version 1.5R3. Cependant, la différence entre le nombre de corrections de bogues des clients instables et des clients stables est significative dans le reste des versions. De plus, les mesures de l'effet varient entre faible et moyen. Selon ces résultats, nous pouvons rejeter l'hypothèse nulle $H_{0_{RQ2}}$ pour Rhino.

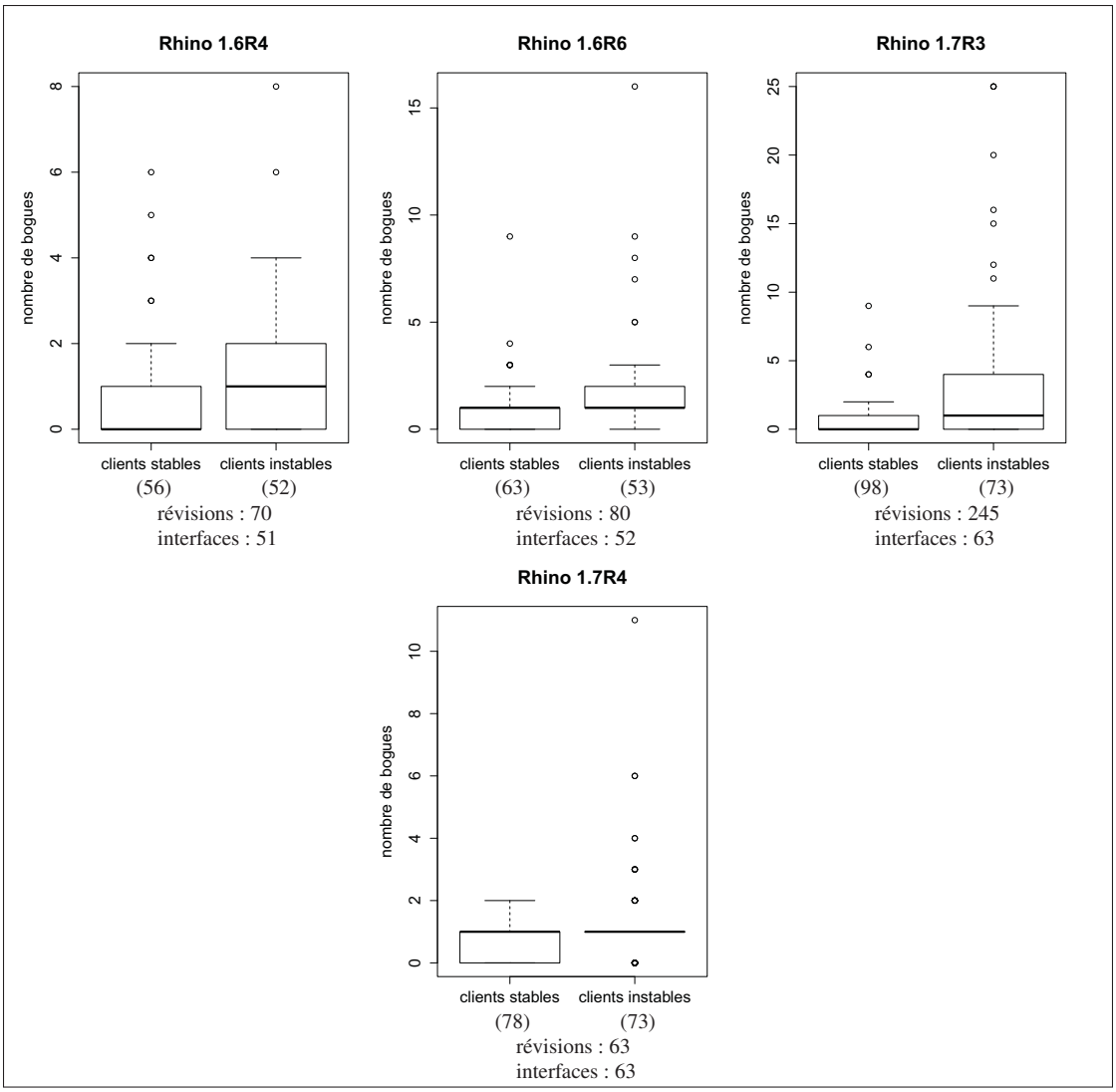

Figure 3.14 Boîtes à moustaches des nombres de corrections de bogues des clients stables et instables dans Rhino (1.6R4 - 1.7R4).

3.3 Discussion des résultats

Les résultats statistiques présentés dans les sections précédentes confirment l'existence d'une différence statistiquement significative entre les clients stables et les clients instables en ce qui concerne leur prédisposition aux changements et aux bogues. Dans la majorité des versions des systèmes étudiés, nous avons trouvé des valeur-p significatives. Les deux hypothèses nulles $H_{0_{RQ1}}$ et $H_{0_{RQ2}}$ ont été rejetées en faveur des hypothèses alternatives H_{a_1} et H_{a_2}. Nous

avons conclu que les clients des interfaces qui ont des couplages avec les implémentations sont plus exposés aux changements et aux bogues dans l'évolution logicielle. Dans cette section, nous discutons les résultats obtenus en ce qui concerne les deux questions de recherche. Nous examinons aussi des cas concrets des deux types des clients à partir des logiciels étudiés.

3.3.1 JHotDraw

RQ1 : couplages instables et prédisposition des clients aux changements

Le test de Mann-Whitney sur les distributions du nombre de changements des clients instables et des clients stables indique qu'il y a une différence significative entre les deux groupes de données dans les deux versions analysées. En examinant les données obtenues concernant les changements des clients dans JHotDraw 5, nous observons que les clients couplés avec les implémentations ont des valeurs plus élevées par rapport aux clients ayant uniquement des couplages avec les interfaces. Par exemple, la classe *DrawApplication* qui joue le rôle de l'éditeur graphique comportant le menu et la palette de dessin a été modifiée 34 fois par les développeurs. Cette dernière est un client de l'interface *Tool* et de ses implémentations *SelectionTool* et *NullTool*. De plus, le point de stabilité *Tool* possède une forte extension avec 29 implémentations dans sa zone de variations. Nous avons trouvé que dans six révisions, le client *DrawApplication* et la classe *SelectionTool* ont subi un changement parallèle. Selon Aversano *et al.* (2007), le co-changement constitue la modification d'un ensemble de classes dans une même révision et par le même développeur. Ce type de changement permet d'analyser l'impact des modifications d'une entité logicielle sur le reste du système et d'étudier aussi la propagation des changements. Dans notre étude, nous n'avons pas étudié l'impact du changement des implémentations sur les clients. Nous planifions d'étudier et d'analyser le co-changement dans nos travaux futurs.

De plus, *DrawApplication* joue aussi le rôle du client de l'interface *Command* qui possède dans sa zone de variabilité 20 commandes concrètes. Les classes *CommandChoice, CommandButton* et *CommandMenu* sont des clients stables de l'interface *Command*. Toutefois la moyenne du

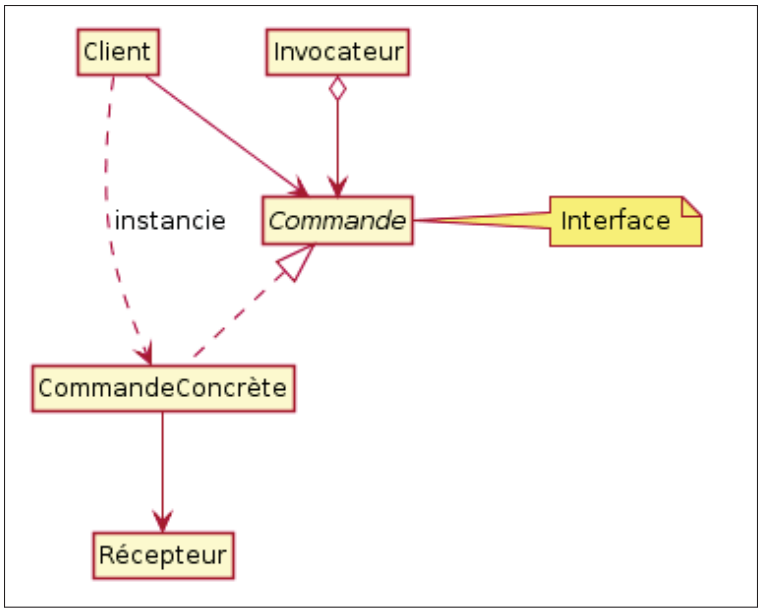

Figure 3.15 Diagramme de classes du patron de conception Commande tiré de (Gamma *et al.*, 1994).

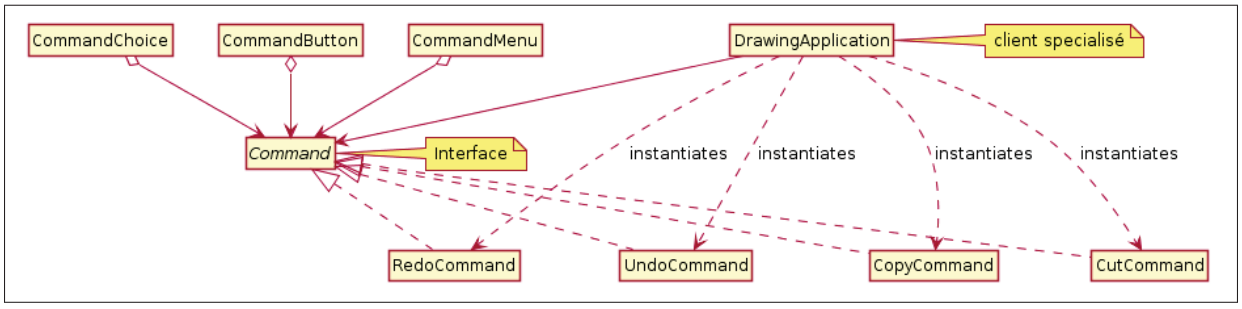

Figure 3.16 Exemple du client spécialisé dans le patron Commande du projet JHotDraw 5.

nombre de changements de ces clients est égale à cinq, cette valeur est totalement inférieure au nombre de changements du client instable *DrawApplication*.

En présentant le diagramme de classes du patron Commande dans la Figure 3.15, nous constatons que le client de l'interface possède du couplage avec l'implémentation. Les auteurs des patrons GoF ont ajouté une dépendance entre le client et l'implémentation. Le client de l'interface a besoin de créer une instance de la commande concrète. Nous avons considéré ces clients comme des clients spécialisés parce que leur dépendance avec l'implémentation est cohérente

selon la solution du patron Commande. Dans la Figure 3.16, le client DrawApplication joue le rôle d'un client spécialisé parce qu'il possède des couplages avec les implémentations qui sont cohérents avec le patron Commande présenté dans la Figure 3.15.

Nous avons aussi trouvé que l'interface *StorageFormat* est implémentée par quatre classes. Cette dernière a un client stable *StorageFormatManager* et un client instable qui est la classe *DrawApplication*. Or, le nombre de changements de la classe *StorageFormatManager* dans l'évolution du JHotDraw 5 est égal à sept. D'où la différence entre les deux valeurs est importante. La classe *DrawApplication* possède en totale des couplages instables avec 27 implémentations qui présentent des extensions de six interfaces. Cette classe comporte le plus grand nombre de changements parmi tous les clients identifiés dans le système JHotDraw 5.

Le deuxième exemple concerne la classe *StandardDrawingView*. Elle joue le rôle de l'interface graphique du dessin permettant aux utilisateurs de dessiner les formes géométriques. *StandardDrawingView* a subi 32 changements et elle constitue la deuxième classe client la plus modifiée. Elle possède des couplages instables avec sept implémentations appartenant à trois interfaces différentes. Finalement, la classe *JavaDrawApp* a des couplages instables avec les extensions de cinq interfaces. Cette classe a été modifiée 17 fois par les développeurs dans l'évolution du JHotDraw 5.

D'autre part, les clients stables identifiés dans cette version possèdent des valeurs de changements inférieures à celles des clients instables. La classe *MDIDesktopPane* possède un couplage stable avec l'interface *DesktopListener*. Elle a subi 15 changements et qui constitue la plus grande valeur parmi tous les clients stables.

Nous constatons d'après ces cas que les dépendances avec les implémentations ont un impact sur la prédisposition des clients aux changements dans l'évolution de la cinquième version de JHotDraw. Les interfaces jouent un rôle très important dans le masquage des variations qui sont souvent sujettes aux modifications. Ce mécanisme favorise la stabilité des clients externes en termes d'exposition aux changements lorsque ces derniers ne possèdent pas des couplages avec les implémentations.

Dans la version JHotDraw 7, les résultats statistiques ont montré une différence largement significative en ce qui concerne la première question de recherche. Les clients instables sont plus sujets aux modifications par rapport aux clients stables. Par exemple, la classe *SVGView* joue le rôle de la vue principale du dessin d'images vectorielles en format SVG. Cette dernière possède des couplages vers les implémentations de trois interfaces. Le client *SVGView* a été modifié dans 38 révisions. Cette valeur est supérieure à toutes les valeurs des nombres de changements des clients stables. De plus, les classes instables *DefaultDrawingView* et *MoveHandle* sont parmi les clients les plus modifiés dans JHotDraw 7.

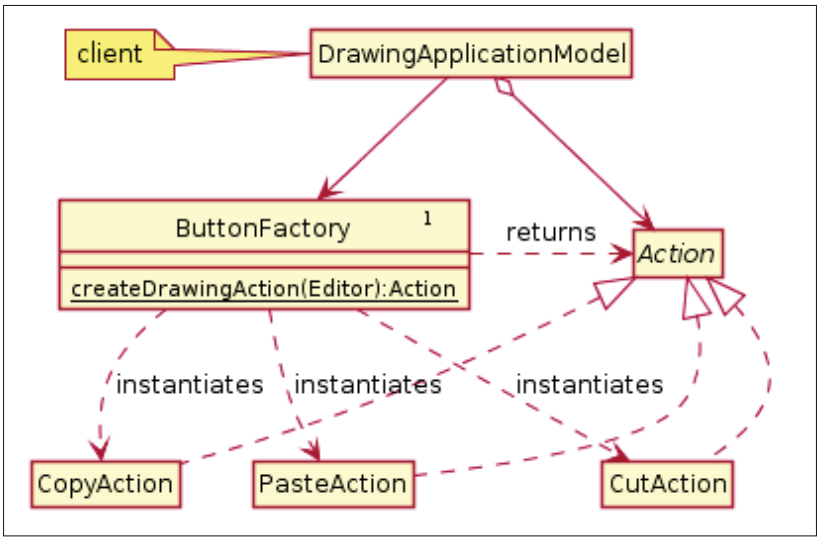

Figure 3.17 Fabrique simple dans le système JHotDraw permettant de retourner les instanciations de l'interface *Action*.

D'autre part, la classe *ButtonFactory* est le client possédant le plus grand nombre de changements parmi tous les clients identifiés. Cette classe a été modifiée dans 50 révisions dans l'évolution du JHotDraw 7. Le rôle principal de cette dernière est une fabrique simple (Figure 3.17) permettant de créer des objets relatifs aux extensions de plusieurs interfaces telles que *Action, Tool, LineDecoration* et *Disposable*. Nous avons évoqué dans la liste des bonnes pratiques du mécanisme d'interface les patrons de fabrication comme une solution pour limiter le couplage créationnel entre les clients et les implémentations. Or, nous constatons dans ce cas que la fabrique limite la propagation des changements aux clients. Lorsqu'il y a de nouvelles

extensions d'une interface, le concepteur apporte un changement seulement dans la fabrique au lieu de modifier tous les clients qui auront besoin des implémentations ajoutées. Les patrons fabriques sont largement utilisés dans le développement logiciel pour traiter les problèmes de la création des objets et pour réduire le couplage risqué entre les clients et les zones de variations des interfaces.

RQ2 : couplages instables et prédisposition des clients aux bogues

Dans cette étude, nous n'avons pas trouvé des résultats statistiques significatifs en ce qui concerne la prédisposition des clients instables aux bogues dans la version JHotDraw 5. La différence n'est pas considérable entre le nombre de corrections de bogues des deux types de client. Ceci est principalement dû au nombre limité des révisions dans lesquelles les développeurs ont corrigé des bogues. Nous avons seulement identifié 23 révisions de correction de bogues sur un total de 177 révisions dans JHotDraw 5. Dans les premières phases de développement, les développeurs se concentrent sur les fonctionnalités. Les défauts sont souvent identifiés après les versions initiales des logiciels.

Les résultats statistiques présentés ont montré une différence légèrement significative en ce qui concerne la prédisposition des clients aux bogues dans JHotDraw 7. En examinant les données des clients, nous constatons que les classes *SVGInputFormat* et *DefaultDrawingView* sont les plus modifiées parmi les clients instables avec neuf corrections de bogues. En outre, la classe *AbstractApplication* constitue le client stable le plus modifié avec dix changements dans le même contexte.

Nous concluons pour le système JHotDraw que le couplage vers les implémentations affecte la prédisposition des clients à l'augmentation du nombre de changements et surtout lorsqu'ils sont couplés avec plusieurs implémentations supposées être masquées par une interface. Cependant, nous n'avons pas trouvé une différence significative entre les clients instables et les clients stables en ce qui concerne la prédisposition de ces derniers aux bogues dans JHotDraw 5. Cette différence est légèrement significative dans la version JHotDraw 7.

3.3.2 ArgoUML

RQ1 : couplages instables et prédisposition des clients aux changements

Les tests statistiques appliqués sur les données collectées à partir du système ArgoUML ont donné des résultats significatifs. Dans la version 0.10, nous observons que la classe *Project-Browser* constitue le client le plus modifié qui comprend 33 changements. Cette dernière est un client instable et joue le rôle de la fenêtre principale de l'application. *ProjectBrowser* invoque les implémentations de l'interface *QuadrantPanel*. Nous avons aussi trouvé dans l'évolution qu'il existe quatre révisions dans lesquelles une des extensions de cette interface et le client *ProjectBrowser* ont été modifiés en parallèle. Ceci veut dire que parfois, les changements se propagent lorsque les clients ne sont pas protégés.

En outre, la classe *Actions* qui gère les différentes actions pouvant être effectuées par l'utilisateur constitue aussi un client instable en invoquant les implémentations de l'interface *Poster*. Cette classe a été modifiée dans 24 révisions. Nous constatons aussi que la différence entre les deux distributions de données est importante. 75% des clients stables ont des taux de changement inférieurs à quatre, alors que 75% des clients instables ont été modifiés dans au moins quatre révisions.

Dans la version 0.12, les clients instables ont aussi des valeurs de changement plus élevées que les clients ayant seulement des couplages avec les interfaces. Les classes *ParserDispaly* et *ProjectBrowser* sont des clients instables et possèdent respectivement 24 et 16 changements dans l'évolution de cette version. Cependant, dans la liste des clients stables, la classe *UML-TextField* a été modifiée dans dix révisions. Le reste des valeurs de changement dans cette liste se situe entre zéro et trois pour la majorité des clients stables.

Dans la version 0.14, nous observons que la moitié des clients instables ont des valeurs de changements supérieures à neuf. La valeur maximale dans cette distribution est relative au client instable *ProjectBrowser* qui a subi 58 changements. Par contre, dans la distribution des

nombres de changements des clients stables, la majorité des valeurs se trouve dans l'intervalle [0,3].

Dans la version 0.18, la différence entre les deux types du couplage est aussi importante. Il existe environ 75% des clients stables ayant des valeurs de modification inférieure à quatre. Cependant, 75% des clients instables ont été modifiés dans au moins dix révisions.
Les résultats sont similaires dans le reste des versions étudiées. Le taux de changements des clients instables dans l'évolution de ArgoUML est supérieur au taux de changements des clients ayant uniquement des couplages stables. Nous constatons que les dépendances avec les éléments des zones de variabilité des interfaces peuvent exposer les classes clients à l'augmentation de leur fréquence de modification, et aussi aux changements propagés par les implémentations.

RQ2 : couplages instables et prédisposition des clients aux bogues

Les résultats statistiques relatifs à la prédisposition des clients instables et stables aux bogues sont aussi significatifs dans toutes les versions explorées de ArgoUML. Dans la version 0.10, la classe *Main* est le client le plus modifié dans un contexte de correction de bogues. Certes que cette dernière possède plusieurs couplages instables créationnels, mais elle joue aussi un rôle important dans la conception de ArgoUML. Ce qui la rend plus exposée aux changements. D'autre part, la classe *ProjectBrowser* comporte un nombre considérable de modifications dans le même contexte. Elle représente la fenêtre principale de l'outil ArgoUML. Cette classe fait appel à plusieurs implémentations des panneaux tels que *NavigatorPane, DetailsPane, MultiEditorPane*, etc. Or, en examinant les révisions de correction de bogues dans lesquelles la classe *ProjectBrowser* a été modifiée, nous observons que la cause du changement dans plusieurs révisions est liée au cadre de l'application et les différentes dispositions des panneaux.

Dans ArgoUML 0.14, la différence statistique est largement significative entre les deux types des clients. En observant les données de l'évolution de cette version, nous trouvons que les classes *ProjectBrowser* et *ParserDisplay* sont parmi les clients instables les plus impliqués dans les révisions de correction de bogues. De plus, nous constatons que 75% des clients stables ont

des valeurs de changements inférieures à quatre, tandis que pour les clients instables, environ 75% de ces classes ont été modifiés dans au moins quatre révisions.

Dans la version 0.18, la différence est aussi importante, et la valeur de delta de Cliff en ce qui concerne la taille d'effet est large (0,83). Environ 75% des clients stables possèdent un nombre de changements inférieur à quatre. Alors que 75% des clients instables ont subi plus que six modifications dans un contexte de correction de bogues.

En général, les validations statistiques ont donné des résultats significatifs dans les versions analysées du système ArgoUML. Nous avons trouvé une corrélation entre le couplage avec les implémentations des interfaces et la prédisposition des clients aux changements et aux bogues dans l'évolution logicielle.

3.3.3 Rhino

RQ1 : couplages instables et prédisposition des clients aux changements

Selon les résultats obtenus, nous avons trouvé une différence statistiquement significative entre les différents de clients en ce qui concerne les changements dans les versions du projet Rhino. Les classes clients qui invoquent les implémentations des interfaces possèdent des taux de changements plus importants par rapport aux clients qui ont seulement des couplages avec les interfaces.

Dans la version Rhino 1.5R1, nous observons que la classe *Context* constitue le client ayant le plus de changement par rapport à tous les clients identifiés vu qu'elle a été modifiée dans 37 révisions. Cette dernière possède aussi des couplages avec les variations de quatre interfaces. De plus, en examinant les données de l'évolution, nous constatons que dans dix révisions, le client *Context* a subi un changement en parallèle avec au moins une implémentation des interfaces dont elle possède des couplages. Ce type de changement confirme que les couplages risqués avec les variations peuvent souvent faire l'objet de la propagation des modifications aux clients.

D'autre part, la classe *Interpreter* qui joue le rôle de l'interpréteur principal de l'application a été modifiée dans 30 révisions. Cette classe est un client de l'interface *Scriptable* et possède aussi des couplages avec les implémentations de cette dernière. Cependant, nous trouvons dans l'évolution de la version Rhino 1.5R1 que le client *Interpreter* a été modifié dans sept révisions où les implémentations de l'interface Scriptable ont subi aussi des changements. Nous confirmons une autre fois que ce type de couplage peut introduire la propagation des changements entre les variations et les clients d'une interface.

Dans la version Rhino 1.5R2, les résultats sont similaires concernant la différence des clients instables et des clients stables en ce qui concerne leur prédisposition aux changements. Environ 50% des clients instables ont été modifiés dans au moins sept révisions, tandis que la majorité des clients stables possèdent des valeurs de changement entre zéro et cinq. Les classes clients les plus modifiées dans cette version sont les clients instables *Context, Interpreter* et *Codegen* ayant respectivement 33, 22 et 19 changements.

Dans la version Rhino 1.5R4.1, nous constatons que le client *Codegen* a des couplages avec les implémentations de l'interface *Scriptable*. En examinant l'évolution des changements de cette version, nous observons que dans dix révisions, les changements ont été effectués en même temps sur le client *Codegen* et les implémentations de l'interface.

RQ2 : couplages instables et prédisposition des clients aux bogues

La différence en ce qui concerne le lien entre les deux types des clients et leur prédisposition aux bogues est aussi significative dans toutes les versions étudiées à l'exception de la version Rhino 1.5R3. En inspectant les données de l'évolution de cette dernière, nous constatons que le nombre des révisions de correction de bogues est faible. D'autre part, nous constatons que dans les versions 1.5R1 et 1.5R2, les clients stables ont des faibles valeurs de changements dont la majorité est inférieure à deux. Tandis que 50% des clients instables ont subi un changement dans au moins une révision. Nous concluons que dans le projet Rhino, les clients des interfaces ayant des dépendances avec les implémentations sont plus prédisposés aux changements et aux

bogues que les clients ayant seulement des couplages avec l'interface. La bonne utilisation de ce mécanisme peut réduire l'impact des changements au niveau des variations sur le reste du système.

3.4 Obstacles à la validité

Dans notre étude empirique, nous avons montré que le couplage aux implémentations des interfaces affecte la prédisposition des clients aux changements et aux bogues. Cette validation empirique supporte l'idée que les abstractions sont utiles dans la conception logicielle en dissimulant les artefacts instables et en diminuant les risques de la propagation des changements aux clients. Cependant, nous ne confirmons pas que les dépendances des classes clients avec les implémentations sont la cause principale d'introduction des changements et des défauts dans les classes clients. Dans cette section, nous analysons les obstacles à la validité de notre étude en ce qui concerne la validité externe, interne et de construit.

3.4.1 Validité externe

La validité externe concerne la généralisabilité de nos résultats. Dans cette étude, nous avons analysé trois systèmes libres qui varient selon le domaine d'application, la taille, le nombre de révisions des versions analysées et leur durée du développement. Néanmoins, nous suggérons que des études additionnelles soient effectuées en tenant compte d'autres langages de programmation orientée objet, ainsi que l'analyse d'autres mécanismes d'abstraction.

3.4.2 Validité interne

Les obstacles à la validité interne n'affectent pas notre approche puisque nous avons réalisé une étude exploratoire. Cette validité est pertinente dans les études qui cherchent à établir une relation de causalité. Or, nous ne cherchons pas à prouver que le couplage vers les implémentations est la cause de la prédisposition des clients aux changements.

3.4.3 Validité de construit

La validité de construit se réfère à la pertinence de mesures et concerne la relation entre la théorie et l'observation. Nous avons utilisé l'outil IHvis pour la détection des clients des interfaces dans les systèmes étudiés. Rufiange et Fuhrman (2014) ont présenté cet outil dans leur approche pour la visualisation de l'évolution des zones de variations. Les auteurs ont testé IHvis en effectuant une étude sur des participants afin d'analyser l'efficacité de l'approche. Toutefois, ils n'ont pas vérifié la précision de IHvis en ce qui concerne l'extraction de données à partir des dépôts du code source. L'exactitude de IHvis peut affecter les résultats obtenus parce que nous pouvons ajouter des classes dans la liste des clients instables tandis que ces dernières ne possèdent pas des couplages avec les implémentations et vice-versa.

Nous avons utilisé JGit pour la détection des changements au niveau des clients. Cet outil analyse les fichiers journaux de changement dans les dépôts du code source. L'identification des changements est fiable parce que nous avons considéré un changement d'une classe lorsque cette dernière a subi n'importe quel type de modification dans l'évolution logicielle. Cependant, nous avons appliqué des heuristiques et des expressions rationnelles pour la détection des révisions de correction de bogues. L'analyse prend en considération le message de la révision afin de déterminer si cette dernière est un changement dans un contexte de bogue. Parfois, les développeurs ne précisent pas dans le message d'une révision qu'il s'agit d'une correction de bogues, ou ils oublient d'ajouter les identifiants de la documentation des défauts dans les systèmes de suivi des problèmes. Nous avons réalisé des vérifications manuelles lorsque les messages des révisions manquent des informations. Toutefois, il peut y avoir des révisions de correction de bogues que nous n'avons pas identifiées parce qu'elles ne comprennent pas les mots clés utilisés dans l'analyse de message.

3.5 Limites de l'approche

Dans notre étude, nous avons analysé les couplages avec les implémentations des interfaces et leur impact sur la prédisposition des clients aux changements et aux bogues. L'approche se

limite seulement aux systèmes développés en Java, tandis que d'autres langages de programmation orientée objet tels que C++ peuvent être envisagés. De plus, les dépôts du code source des projets étudiés sont en format SVN. Nous avons utilisé l'outil SubGit pour la migration des projets Git vers SVN. En outre, nous avons étudié seulement les interfaces Java comme des points de stabilité permettant de cacher les variations. Cependant, il existe d'autres mécanismes d'abstraction tels que les classes abstraites ou les classes avec les méthodes virtuelles en C++. D'autres méthodes peuvent être utilisées pour définir des abstractions. Par exemple, une classe façade peut dissimuler un ensemble de code complexe et facilite leur utilisation aux clients. Cette approche se limite aussi à l'analyse des changements au niveau des clients en tenant compte de n'importe quel type de modification. L'étude des types de changement (par exemple, changement structurel ou non structurel) peut donner un aperçu sur les modifications les plus effectuées sur les clients des interfaces. De plus, nous avons trouvé des cas où les changements dans les variations engendrent des modifications au niveau des clients instables. Les couplages risqués peuvent introduire la propagation des changements. Néanmoins, nous avons considéré seulement la prédisposition des clients à toute sorte de changement.

CONCLUSION

Le masquage d'information est un mécanisme de base dans le développement moderne. L'isolation des artefacts instables qui représentent des décisions de conception difficiles à gérer au fils du temps facilite le développement en équipe, minimise les couplages entre les modules logiciels et réduit l'impact des changements au niveau des détails sur le reste du système. Les interfaces constituent des techniques de masquage d'information largement utilisées dans la programmation orientée objet en Java. Les implémentations cachées derrière l'interface sont les parties complexes des systèmes souvent exposées aux changements dans l'évolution logicielle. La protection des variations dissimulées ne peut être achevée que si les clients dépendent seulement de l'interface. La bonne utilisation de ce mécanisme facilite la compréhension et la maintenabilité des logiciels en favorisant la programmation modulaire et la minimisation de la propagation des changements provoquée par les implémentations. Cependant, la conception des interfaces n'est pas une tâche triviale et leur mauvaise application peut être coûteuse.

Dans ce mémoire, nous avons proposé une étude empirique visant à évaluer l'impact de l'existence des couplages entre les clients et les implémentations sur l'introduction de changements dans l'évolution des logiciels. Nous avons analysé la différence entre les clients stables et les clients instables en ce qui concerne leur prédisposition aux changements et aux bogues. Pour répondre à nos questions de recherche, nous avons suivi une démarche de collecte et d'analyse de données extraites à partir de trois systèmes largement étudiés dans la littérature. De plus, nous avons utilisé des outils pour identifier les clients des interfaces et pour détecter les changements dans l'évolution. Nous avons aussi appliqué des expressions rationnelles et des heuristiques afin d'identifier les révisions de correction de bogues.

Les résultats statistiques obtenus ont montré qu'il existe une différence significative entre les clients instables et les clients stables en ce qui concerne leur prédisposition aux changements, et cela pour presque la majorité des versions des systèmes analysés. Nous avons observé à travers nos deux questions de recherche que les classes clients possédant des dépendances avec les implémentations sont plus exposées aux changements et ils sont plus impliqués dans les révisions de correction des bogues. D'autre part, nous avons trouvé des cas où les modifications

au niveau des variations peuvent faire l'objet de la propagation des changements aux clients. Ainsi que le nombre de couplages avec les détails des interfaces est corrélé avec la fréquence de changement des classes clients.

Les résultats de notre recherche ont indiqué que l'introduction des dépendances vers les implémentations par les développeurs peut affecter le coût de changement et provoque aussi les bogues dans les systèmes. Cette étude a davantage révélé la pertinence du principe de masquage d'information dans l'appui des structures de conception qui supporte le faible couplage avec les éléments potentiellement instables.

Suite à cette contribution, de nouvelles pistes de recherches peuvent être envisagées dans le futur. Les chercheurs intéressés par les approches empiriques peuvent étudier d'autres mécanismes de masquage d'information tels que les classes abstraites, les classes avec des méthodes virtuelles en C++ ou les classes intermédiaires (par exemple, Façade dans GoF (Gamma *et al.*, 1994)). D'autre part, l'étude de l'impact des couplages risqués sur la propagation de changements est envisageable. Nous avons constaté dans notre analyse que parfois les changements au niveau des variations peuvent se propager aux clients des interfaces. Nous avons aussi trouvé des cas où des clients spécialisés possèdent des couplages avec les implémentations et qu'il existe une cohérence avec la structure des patrons tels que Commande, Décorateur, et Composite dans le logiciel JHotDraw. Ce type de client peut ne pas représenter un risque sur la stabilité des logiciels. L'étude des clients spécialisés est envisageable dans les travaux futurs. Enfin, les chercheurs peuvent aussi analyser l'impact des techniques promouvant la bonne conception des interfaces (par exemple, l'injection de dépendances, les patrons de fabrication) sur la minimisation des couplages et le coût de changement.

Ce travail a été présenté dans une affiche lors du consortium pour la recherche en génie logiciel (Consortium for Software Engineering Research Spring 01-06-2016) tenu à l'université d'Alberta à Edmonton. Nous visons maintenant à publier notre contribution dans un journal scientifique.

BIBLIOGRAPHIE

Abdeen, H., H. Sahraoui, et O. Shata. 2013a. « How we design interfaces, and how to assess it ». In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. p. 80–89. IEEE.

Abdeen, H., O. Shata, et A. Erradi. 2013b. « Software interfaces : On the impact of interface design anomalies ». In *Computer Science and Information Technology (CSIT), 2013 5th International Conference on*. p. 184–193. IEEE.

Alghamdi, F. M. et M. R. J. Qureshi. 2014. « Impact of Design Patterns on Software Maintainability ». *International Journal of Intelligent Systems and Applications*, vol. 6, n° 10, p. 41.

Alshayeb, M., M. Naji, M. O. Elish, et J. Al-Ghamdi. August 2011. « Towards measuring object-oriented class stability ». *IET Software*, vol. 5, n° 4, p. 415-424.

Ampatzoglou, A., A. Chatzigeorgiou, S. Charalampidou, et P. Avgeriou. 2015. « The Effect of GoF Design Patterns on Stability : A Case Study ». *Software Engineering, IEEE Transactions on*, vol. 41, n° 8, p. 781-802.

Ampatzoglou, A., A. Kritikos, E.-M. Arvanitou, A. Gortzis, F. Chatziasimidis, et I. Stamelos. 2011. « An empirical investigation on the impact of design pattern application on computer game defects ». In *Proceedings of the 15th International Academic MindTrek Conference : Envisioning Future Media Environments*. p. 214–221. ACM.

Ampatzoglou, A., G. Frantzeskou, et I. Stamelos. 2012. « A methodology to assess the impact of design patterns on software quality ». *Information and Software Technology*, vol. 54, n° 4, p. 331–346.

Ampatzoglou, A., S. Charalampidou, et I. Stamelos. 2013. « Research state of the art on GoF design patterns : A mapping study ». *Journal of Systems and Software*, vol. 86, n° 7, p. 1945–1964.

Arvanitou, E.-M., A. Ampatzoglou, A. Chatzigeorgiou, et P. Avgeriou. 2015. « Introducing a Ripple Effect Measure : A Theoretical and Empirical Validation ». In *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*. p. 1–10. IEEE.

Aversano, L., G. Canfora, L. Cerulo, C. Del Grosso, et M. Di Penta. 2007. « An empirical study on the evolution of design patterns ». In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. p. 385–394. ACM.

Bandi, R. K., V. K. Vaishnavi, et D. E. Turk. 2003. « Predicting maintenance performance using object-oriented design complexity metrics ». *Software Engineering, IEEE Transactions on*, vol. 29, n° 1, p. 77–87.

Bansiya, J. 2000. « Evaluating Framework Architecture Structural Stability ». *ACM Comput. Surv.*, vol. 32, n° 1es.

Bieman, J. M., D. Jain, et H. J. Yang. 2001. « OO design patterns, design structure, and program changes : an industrial case study ». In *Software Maintenance, 2001. Proceedings. IEEE International Conference on*. p. 580–589. IEEE.

Bieman, J. M., G. Straw, H. Wang, P. W. Munger, et R. T. Alexander. 2003. « Design patterns and change proneness : An examination of five evolving systems ». In *Software metrics symposium, 2003. Proceedings. Ninth international*. p. 40–49. IEEE.

Boxall, M. A. et S. Araban. 2004. « Interface metrics for reusability analysis of components ». In *Software Engineering Conference, 2004. Proceedings. 2004 Australian*. p. 40–51. IEEE.

Brown, W. H., R. C. Malveau, et T. J. Mowbray. 1998. « AntiPatterns : refactoring software, architectures, and projects in crisis ». *Wiley*.

Cataldo, M., C. R. De Souza, D. L. Bentolila, T. C. Miranda, et S. Nambiar. 2010. « The impact of interface complexity on failures : an empirical analysis and implications for tool design ». *School of Computer Science, Carnegie Mellon University, Tech. Rep*.

Cliff, N. 1993. « Dominance statistics : Ordinal analyses to answer ordinal questions. ». *Psychological Bulletin*, vol. 114, n° 3, p. 494.

Cockburn, A. 1996. « The interaction of social issues and software architecture ». *Communications of the ACM*, vol. 39, n° 10, p. 40–46.

Di Penta, M., L. Cerulo, Y.-G. Guéhéneuc, et G. Antoniol. 2008. « An empirical study of the relationships between design pattern roles and class change proneness ». In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. p. 217–226. IEEE.

Eaddy, M., T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, et A. V. Aho. 2008. « Do crosscutting concerns cause defects ? ». *Software Engineering, IEEE Transactions on*, vol. 34, n° 4, p. 497–515.

Elish, M. 2006. « Do Structural Design Patterns Promote Design Stability ? ». In *Computer Software and Applications Conference, 2006. COMPSAC '06. 30th Annual International*. p. 215-220. IEEE.

Elish, M. O. et D. Rine. 2003. « Investigation of metrics for object-oriented design logical stability ». In *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*. p. 193–200. IEEE.

Fayad, M. E. et A. Altman. 2001. « Thinking objectively : an introduction to software stability ». *Communications of the ACM*, vol. 44, n° 9, p. 95.

Fowler, M. 2004. « Inversion of control containers and the dependency injection pattern ». In *URL : http ://martinfowler.com/articles/injection.html*.

Gall, H., M. Jazayeri, et J. Krajewski. 2003. « CVS release history data for detecting logical couplings ». In *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*. p. 13–23. IEEE.

Gamma, E., R. Helm, R. Johnson, et J. Vlissides. 1994. « Design Patterns : Elements of Object-Oriented Software Architecture ». *Addison-Wesley*, vol. 9, p. 12.

Gatrell, M. et S. Counsell. 2011. « Design patterns and fault-proneness a study of commercial C# software ». In *Research Challenges in Information Science (RCIS), 2011 Fifth International Conference on*. p. 1–8. IEEE.

Gatrell, M., S. Counsell, et T. Hall. 2009. « Design patterns and change proneness : a replication using proprietary C# software ». In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. p. 160–164. IEEE.

Grosser, D., H. A. Sahraoui, et P. Valtchev. 2002. « Predicting software stability using case-based reasoning ». In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*. p. 295–298. IEEE.

Grosser, D., H. A. Sahraoui, et P. Valtchev. 2003. « An analogy-based approach for predicting design stability of Java classes ». In *Software Metrics Symposium, 2003. Proceedings. Ninth International*. p. 252–262. IEEE.

Guéhéneuc, Y.-G. et G. Antoniol. 2008. « Demima : A multilayered approach for design pattern identification ». *Software Engineering, IEEE Transactions on*, vol. 34, n° 5, p. 667–684.

Hassan, Y. S., 2007. *Measuring software architectural stability using retrospective analysis*. King Fahd University of Petroleum and Minerals : Proquest.

Henning, M. 2007. « API Design Matters ». *Queue, ACM*, vol. 5, n° 4, p. 24–36.

ISO-1926. 2001. « IEC 9126-1 : Software Engineering-Product Quality-Part 1 : Quality Model ». *Geneva, Switzerland : International Organization for Standardization*, p. 27.

Izurieta, C. et J. M. Bieman. 2013. « A multiple case study of design pattern decay, grime, and rot in evolving software systems ». *Software Quality Journal*, vol. 21, n° 2, p. 289–323.

Jaafar, F., Y.-G. Guéhéneuc, S. Hamel, et G. Antoniol. 2014a. « Detecting asynchrony and dephase change patterns by mining software repositories ». *Journal of Software : Evolution and Process*, vol. 26, n° 1, p. 77–106.

Jaafar, F., F. Khomh, Y.-G. Guéhéneuc, et M. Zulkernine. 2014b. « Anti-pattern Mutations and Fault-proneness ». In *Quality Software (QSIC), 2014 14th International Conference on*. p. 246–255. IEEE.

Jazayeri, M., 2002. *On Architectural Stability and Evolution*. Ada-Europe '02. London, UK, UK : Springer-Verlag, 13–23 p.

Juristo, N. et S. Vegas. 2011. « Design patterns in software maintenance : an experiment replication at UPM-experiences with the RESER'11 joint replication project ». In *Replication in Empirical Software Engineering Research (RESER), 2011 Second International Workshop on*. p. 7–14. IEEE.

Kelly, D. 2006. « A study of design characteristics in evolving software using stability as a criterion ». *Software Engineering, IEEE Transactions on*, vol. 32, n° 5, p. 315–329.

Khomh, F., Y.-G. Guéhéneuc, et G. Antoniol. 2009a. « Playing roles in design patterns : An empirical descriptive and analytic study ». In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. p. 83–92. IEEE.

Khomh, F., M. D. Penta, et Y.-G. Gueheneuc. 2009b. « An exploratory study of the impact of code smells on software change-proneness ». In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. p. 75–84. IEEE.

Khomh, F., M. Di Penta, Y.-G. Guéhéneuc, et G. Antoniol. 2012. « An exploratory study of the impact of antipatterns on class change-and fault-proneness ». *Empirical Software Engineering*, vol. 17, n° 3, p. 243–275.

Krein, J. L., L. J. Pratt, A. B. Swenson, A. C. MacLean, C. D. Knutson, et D. L. Eggett. 2011. « Design patterns in software maintenance : An experiment replication at Brigham Young University ». In *Replication in Empirical Software Engineering Research (RESER), 2011 Second International Workshop on*. p. 25–34. IEEE.

Larman, C. 2001. « Protected variation : The importance of being closed ». *Software, IEEE*, vol. 18, n° 3, p. 89–91.

Larman, C., 2005. *Applying UML and patterns : an introduction to object-oriented analysis and design and iterative development*. éd. 3rd. Upper Saddle River, NJ : Prentice Hall PTR.

Lehman, M. M. 1980. « Programs, life cycles, and laws of software evolution ». *Proceedings of the IEEE*, vol. 68, n° 9, p. 1060–1076.

Li, W., L. Etzkorn, C. Davis, et J. Talburt. 2000. « An empirical study of object-oriented system evolution ». *Information and Software Technology*, vol. 42, n° 6, p. 373–381.

Mann, H. B. et D. R. Whitney. 1947. « On a test of whether one of two random variables is stochastically larger than the other ». *The annals of mathematical statistics*, p. 50–60.

Martin, R. 1997. *Stability–c++ report*. Technical report.

Martin, R. C. 2000. « Design principles and design patterns ». *Object Mentor*, vol. 1, p. 34.

McConnell, S., 2004. *Code Complete, Second Edition*. Redmond, WA, USA : Microsoft Press.

McGill, R., J. W. Tukey, et W. A. Larsen. 1978. « Variations of box plots ». *The American Statistician*, vol. 32, n° 1, p. 12–16.

Meyer, B., 1988. *Object-Oriented Software Construction*. éd. 1st. Upper Saddle River, NJ, USA : Prentice-Hall, Inc.

Nanthaamornphong, A. et J. C. Carver. 2011. « Design patterns in software maintenance : An experiment replication at University of Alabama ». In *Replication in Empirical Software Engineering Research (RESER), 2011 Second International Workshop on*. p. 15–24. IEEE.

Olbrich, S., D. S. Cruzes, V. Basili, et N. Zazworka. 2009. « The evolution and impact of code smells : A case study of two open source systems ». In *Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement*. p. 390–400. IEEE Computer Society.

Parnas, D. L. 1972. « On the criteria to be used in decomposing systems into modules ». *Communications of the ACM*, vol. 15, n° 12, p. 1053–1058.

Prechelt, L., B. Unger, W. F. Tichy, P. Brossler, et L. G. Votta. 2001. « A controlled experiment in maintenance : comparing design patterns to simpler solutions ». *IEEE Transactions on Software Engineering,*, vol. 27, n° 12, p. 1134–1144.

Rapu, D., S. Ducasse, T. Gîrba, et R. Marinescu. 2004. « Using history information to improve design flaws detection ». In *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*. p. 223–232. IEEE.

Romano, D. et M. Pinzger. 2011. « Using source code metrics to predict change-prone java interfaces ». In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. p. 303–312. IEEE.

Romano, D., P. Raila, M. Pinzger, et F. Khomh. 2012. « Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes ». In *Reverse Engineering (WCRE), 2012 19th Working Conference on*. p. 437–446. IEEE.

Rufiange, S. et C. P. Fuhrman. 2014. « Visualizing protected variations in evolving software designs ». *Journal of Systems and Software*, vol. 88, p. 231–249.

Śliwerski, J., T. Zimmermann, et A. Zeller. 2005. « When do changes induce fixes ? ». In *ACM sigsoft software engineering notes*. p. 1–5. ACM.

Soong, N. L. 1977. « A program stability measure ». In *Proceedings of the 1977 annual conference*. p. 163–173. ACM.

Tonu, S. A., A. Ashkan, et L. Tahvildari. March 2006. « Evaluating architectural stability using a metric-based approach ». In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*. p. 10 pp.-270.

Tsantalis, N., A. Chatzigeorgiou, G. Stephanides, et S. T. Halkidis. 2006. « Design pattern detection using similarity scoring ». *Software Engineering, IEEE Transactions on*, vol. 32, n° 11, p. 896–909.

Van Gurp, J. et J. Bosch. 2002. « Design erosion : problems and causes ». *Journal of systems and software*, vol. 61, n° 2, p. 105–119.

Vanbrabant, R., 2008. *Google Guice : agile lightweight dependency injection framework*. New York, NY : Apress.

Vokáč, M. 2004. « Defect frequency and design patterns : An empirical study of industrial code ». *Software Engineering, IEEE Transactions on*, vol. 30, n° 12, p. 904–917.

Vokáč, M., W. Tichy, D. I. Sjoberg, E. Arisholm, et M. Aldrin. 2004. « A controlled experiment comparing the maintainability of programs designed with and without design patterns-a replication in a real programming environment ». *Empirical Software Engineering*, vol. 9, n° 3, p. 149–195.

Yau, S. S. et J. S. Collofello. 1980. « Some stability measures for software maintenance ». *Software Engineering, IEEE Transactions on*, , p. 545–552.