UNIVERSITÉ DU QUÉBEC

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

COMME EXIGENCE PARTIELLE DE LA MAÎTRISE EN
MATHÉMATIQUES ET INFORMATIQUE APPLIQUÉES

PAR
JORDANE CURE

ÉVALUATION DE L'IMPACT DES DESIGN PATTERNS SUR LA TESTABILITÉ ET LA CHANGEABILITÉ DES SYSTÉMES ORIENTÉS OBJET.

AOÛT 2016

Résumé

Dans le cycle de vie d'un projet de développement logiciel, effectuer des changements dans le code source du logiciel après sa livraison est une action courante. Cependant, réaliser des changements sur des projets ayant une mauvaise maintenabilité est une tâche complexe, qui prend du temps et qui peut entrainer des erreurs. Ceci a également des conséquences très importantes sur les coûts des logiciels.

Depuis plusieurs années, l'utilisation des design patterns s'est généralisée. Ils sont réputés pour être des solutions fiables et éprouvées.

Considérant les enjeux liés à la maintenabilité, il est intéressant de se demander si les design patterns ont un impact sur la maintenabilité. C'est pourquoi ce travail tente d'évaluer l'impact des design patterns sur deux sous-caractéristiques importantes de la maintenabilité : la testabilité et la changeabilité.

La méthodologie suivie dans ce travail repose sur une étude empirique du code source de l'ensemble des versions de quatre logiciels (pour un total de 161 versions différentes). Cette étude tente d'apporter des éléments de réponse en effectuant des analyses selon deux perspectives. D'une part, en comparant les classes qui sont impliquées dans un design pattern avec les classes qui ne sont pas impliquées dans un design pattern. D'autre part, en comparant les classes avant et après leur implication dans un design pattern. Ces deux pistes d'analyse pourraient permettre de révéler les conséquences, en terme de testabilité et de changeabilité, engendrées par l'introduction des design patterns.

Abstract

In the life cycle of a software development project, modification of the source code of a software after its delivery is a common action. However, making changes on projects with poor maintainability is a complex task, which takes time and can lead to errors. It also has important implications for software costs.

Since several years, the use of design patterns has become widespread. They are reputed to be reliable and proven solutions.

Considering the issue with maintainability, it is interesting to investigate whether design patterns affect maintainability. This work attempts to assess the impact of design patterns on two important sub-characteristics of maintainability / testability and changeability.

The methodology followed in this work is based on an empirical study of the source code of all versions of four programs (for a total of 161 different versions). This study provides answers by performing analyzes from two perspectives. On the one hand, comparing the classes that are involved in a design pattern with classes that are not involved in a design pattern. On the other hand, comparing the classes before and after their involvement in a design pattern. These two tracks of analysis help in revealing the consequences in terms of testability and changeability, caused by the introduction of design patterns.

Remerciements

J'adresse mes remerciements aux personnes qui m'ont aidé dans la réalisation de ce mémoire.

En premier lieu, je remercie mon directeur de recherche, professeur au département de mathématiques et d'informatique de l'Université du Québec à Trois-Rivières, Mourad Badri. En tant que directeur, il m'a guidé dans mon travail et m'a aidé à trouver des solutions pour avancer.

Je remercie aussi Linda Badri, ma co-directrice de recherche et professeure au département de mathématiques et d'informatique de l'Université du Québec à Trois-Rivières, pour m'avoir aidé pendant ce projet de recherche et pour m'avoir poussé à entreprendre une maitrise en génie logiciel.

Mes remerciements s'adressent également à Yann-Gaël Guéhéneuc, professeur au département de génie informatique et génie logiciel de l'École Polytechnique de Montréal. Son aide m'a été précieuse dans la partie technique liée à l'identification des design patterns.

Finalement, je remercie Laurence Delcroix pour son soutien pendant ces trois années de recherche et son travail de correction effectué sur ce mémoire.

Table des matières

Liste des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Contexte

1.1.1 La maintenance

La maintenance est définie par la norme IEEE 1219 [IEEE 98] comme étant la modification d'un logiciel après sa livraison. Ces modifications du logiciel peuvent avoir pour objectif la correction d'erreurs, l'adaptation à un nouvel environnement ou l'amélioration de ses performances et de ses attributs.

L'étape de maintenance est coûteuse dans le cycle de vie des projets informatiques. Selon les études, ces coûts représenteraient entre 50 et 70% du coût total d'un projet de développement logiciel [Somm 01, Lee 00]. Cette proportion est variable selon les projets, car la demande en ressources nécessaires pour effectuer des modifications pendant la phase de maintenance est accrue par plusieurs facteurs tel que [Shir 03] :

- La complexité et la taille.
- La différence des équipes en charge du développement.
- Une mauvaise documentation.
- Une durée importante pendant laquelle le logiciel est maintenu.

Un des facteurs permettant de réduire et de contrôler ces coûts de maintenance est de veiller à ce que le code source du projet ne soit pas trop complexe ou que les composants du logiciel ne soient pas de taille trop importante. On parle alors de maintenabilité du code source. La norme ISO 9126 [ISO 01] précise les attributs du code source qui influent sur sa maintenabilité, à savoir :

La facilité d'analyse :	La facilité d'identification dans le logiciel de l'origine d'un problème.
La stabilité :	Le risque que surviennent des conséquences inattendues suite à une modification.
La testabilité :	L'effort nécessaire pour valider un logiciel.
La changeabilité :	La facilité avec laquelle un logiciel peut être modifié.

Contrôler ces quatre caractéristiques du code permet donc d'obtenir un logiciel maintenable et ainsi réduire les coûts liés à sa phase de maintenance.

1.1.2 Les design patterns

Les "design patterns" ou "patrons de conception" en français, sont des modèles de conception délivrés de l'expérience par des experts. Ils apportent des solutions génériques à des problèmes fréquemment rencontrés lors de la conception. Par exemple, le pattern "observer"[Gamm 95] propose une solution pour les cas où plusieurs classes doivent être informées des changements qui surviennent dans une autre classe. Généralement, cette solution est décrite sous forme de diagramme de classes.

L'utilisation des design patterns permettrait de faciliter la communication entre les développeurs et garantirait une bonne qualité de conception du projet [Ager 98].

1.2 Problématique

Considérant l'importance de la maintenance, il est nécessaire de savoir si cette bonne qualité de conception que permettrait les design patterns, implique une bonne maintenabilité des logiciels. Cependant, jusqu'a présent, très peu d'études ont analysé la relation entre design pattern et maintenabilité.

C'est donc cette absence d'étude qui motive ce travail de recherche. En effet, celui-ci porte sur l'analyse de l'impact des design patterns sur la maintenabilité, en considérant l'effort nécessaire pour valider un logiciel, la testabilité, et la facilité avec laquelle un logiciel peut être modifié, la changeabilité.

1.3 Organisation du mémoire

Ce mémoire est divisé en quatre parties. La première partie est une introduction au concept de design patterns et à la notion de qualité logicielle. L'objectif est de synthétiser les connaissances essentielles à la bonne compréhension du contexte dans lequel s'inscrit ce projet de recherche.

La seconde partie propose une revue de littérature sur trois sujets. Les deux premiers sujets portent sur l'évaluation de la testabilité et sur l'évaluation de la changeabilité. La raison d'être de ces deux revues de la littérature est qu'elles ont permis de construire le cadre d'évaluation de la testabilité et de la changeabilité utilisé dans l'étude empirique de ce projet. Le troisième sujet porte sur les techniques utilisées dans la littérature pour évaluer l'impact des design patterns. Cette revue de la littérature a été effectuée pour répertorier comment d'autres études sont parvenues à évaluer l'impact des design patterns. Ces informations ont ensuite permis de définir la méthodologie utilisée dans ce projet.

La troisième partie présente l'étude expérimentale de ce projet de recherche. En suivant l'approche "Goal Question Metric", les objectifs de recherche, les

questions de recherche et les métriques y sont définis. Ensuite, y est décrit le processus de collecte des données ainsi que les outils utilisés et les logiciels (études de cas) considérés pour l'étude expérimentale. Finalement, les résultats permettant de répondre aux questions de recherche sont considérées.

Finalement, la quatrième et dernière partie, après une discussion sur les biais possibles de cette étude et sur les futurs travaux qu'il serait intéressant de réaliser, apporte une conclusion à ce mémoire.

Première partie

Éléments contextuels

Introduction de la Partie I

L'étude présentée dans ce mémoire fait appel à deux notions importantes en génie logiciel. La première notion est un concept de programmation connu par la majorité des développeurs : les design patterns. Le seconde notion est la qualité logicielle, une des raisons d'être de la science du génie logiciel.

Afin de comprendre les tenants et les aboutissants de cette étude, cette première partie est une introduction aux concepts de design pattern et de qualité logicielle.

Le Chapitre 2 aborde la notion de design pattern et le Chapitre 3 introduit la notion de qualité logicielle.

Chapitre 2

Design patterns

2.1 Présentation du concept de design pattern

Depuis les premiers temps de la programmation, les développeurs furent confrontés à différents problèmes de conception. Une partie de ces problèmes étant récurrente, en utilisant leurs expériences, des groupes de développement mirent au point des solutions génériques à ces problèmes. Ces solutions sont communément appelées design patterns. En français, il existe plusieurs appellations : motif de conception, modèle de conception ou encore patron de conception. Dans ce mémoire, nous utiliserons le terme de design pattern.

2.1.1 Historique

Les premiers design patterns sont proposés dans les années 70 par l'architecte Christopher Alexander dans son livre "A Pattern Language". Ces design patterns sont alors appliqués à des problèmes d'architecture [Alex 77] (il n'est pas ici question d'architecture logicielle ou matérielle, mais de l'architecture de batiments). Puis, ces principes sont repris et appliqués au génie logiciel avec, en 1994, le livre "Design Patterns : Elements of Reusable Object-Oriented Software" [Gamm 95] écrit par le Gang Of Four (plus de détails sur ce sujet sont disponibles dans la section 2.2). Depuis, de nombreuses études et de nombreux travaux ont été réalisés sur le sujet. Beaucoup d'auteurs ont même publié leurs propres design patterns, par exemple les patterns GRASP [Larm 12].

2.1.2 Définition

De nombreuses définitions existent pour définir ce qu'est un design pattern. Le Gang Of Four reprend la définition proposée par Christopher Alexander qui dit que "chaque modèle décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le coeur de la solution de ce problème, d'une façon telle que l'on peut réutiliser cette solution des millions de fois, sans jamais le faire deux fois de la même manière" [Alex 77].

Généralement, une description d'un design pattern comporte les informations suivantes :

1. Le nom du pattern.
2. Le problème que le pattern permet de résoudre.
3. La solution proposée pour résoudre le problème.
4. Les avantages et les inconvénients de cette solution.

2.1.3 Avantages

L'utilisation des design patterns permet d'obtenir trois avantages [Ager 98, Larm 05] :

- Les design patterns offrent des solutions éprouvées et validées par des experts. Leur utilisation permet d'obtenir une bonne qualité de la conception.
- Les design patterns sont connus par un grand nombre de développeurs. Ils permettent donc de faciliter la communication entre les développeurs.
- Les design patterns permettent de faciliter la documentation des conceptions logicielles.

2.2 Le gang of four

Le GOF (acronyme de Gang Of Four) désigne les quatre co-auteurs du premier des livres majeurs sur les design patterns [Gamm 95]. Richard Helm, Erich Gamma (un des fondateurs de l'IDE Eclipse et de l'environnement de test jUnit), Ralph Johnson (un des pionniers du langage Smalltalk) et John Vlissides ont mis en commun leurs expériences pour proposer 23 design patterns. Bien que publiés en 1994, ces 23 patterns restent encore aujourd'hui une référence.

Le GOF classe ces 23 patrons en 3 catégories :

- 5 patterns de construction qui définissent comment créer et configurer les objets : fabrique abstraite, créateur, fabrique, prototype et singleton.
- 7 patterns structuraux qui définissent comment organiser les classes d'un programme : adaptateur, pont, composite, décorateur, façade, poids-mouche et proxy.
- 11 patterns comportementaux qui définissent comment répartir les responsabilités entre les objets : chaîne de responsabilité, commande, interpréteur, itérateur, médiateur, memento, observateur, état, stratégie, méthode et visiteur.

Les quatre sous sections suivantes présentent quatre patterns proposés par le GOF : Observer, Composite, Visitor et Prototype. Il est à noter que ces quatre patterns choisis en illustration sont les quatre patterns considérés par l'étude présentée dans ce mémoire. Le choix de ces quatre patterns a été influencé par l'outil utilisé, permettant de détecter les patterns dans le code source. En effet, ce sont ces quatre patterns qui sont détectés avec la meilleure précision par cet outil. Plus de détails sur ce sujet sont disponibles dans la section 7.3.1.

2.2.1 Le pattern Observer

Il peut exister une classe, dite "subject", dont les valeurs changent régulièrement. Il peut aussi exister d'autres classes, les "observer", qui doivent être informées des changements survenus sur le "subject". Cette relation peut être illustrée par des tableaux qui affichent des statistiques sur la production d'une usine (les observers) et un système qui collecte et stocke ces statistiques (le sujet). Si des données sont ajoutées dans le système, les statistiques sont recalculées, puis les tableaux sont informés que de nouvelles statistiques doivent être affichées.

Le pattern Observer propose une solution pour répondre à ce cas d'utilisation qui minimise le couplage entre un sujet d'observation et les classes qui observent ce sujet. La Figure 2.1 présente un diagramme UML de cette solution.

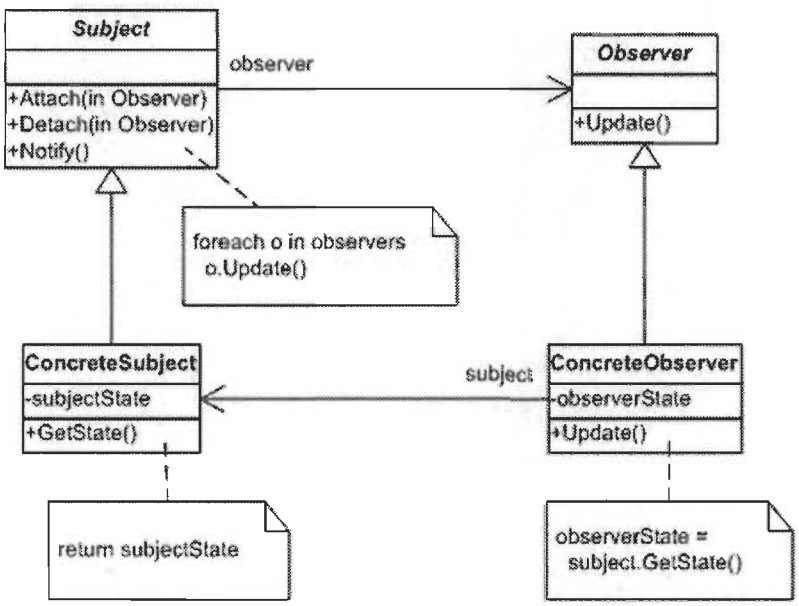

FIGURE 2.1 – Pattern observer [Dofa 15].

2.2.2 Le pattern Composite

Il peut être nécessaire de manipuler des collections composées d'objets simples et d'objets composés d'objets simples. Par exemple, une collection d'articles à vendre dans un magasin : ce magasin vendrait des torchons ("leaf") et des serviettes ("leaf"), mais aussi un kit spécial qui serait un mélange de torchons et de serviettes ("composite"). Le système du magasin ("client") doit pouvoir manipuler les articles ("component") sans avoir à considerer s'il s'agit d'articles simples ou de kits composés d'articles simples.

Le pattern Composite propose une solution pour pouvoir manipuler indifféremment, dans un même ensemble, des torchons, des serviettes ou des kits composés d'un mélange de torchons et de serviettes. La Figure 2.2 présente un diagramme UML de cette solution.

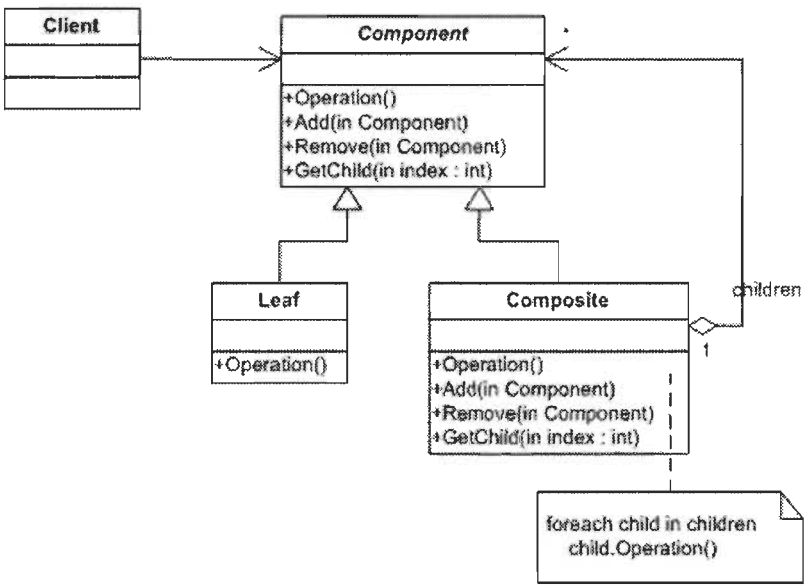

FIGURE 2.2 – Pattern composite [Dofa 15].

2.2.3 Le pattern Visitor

Il est parfois utile de pouvoir séparer un algorithme de la structure d'objets sur laquelle il s'applique. Par exemple, une image en trois dimensions ("ObjectStructure") est composée d'éléments tels que des sphères ("Element") et des polygones ("Element"). Il peut être nécessaire de visiter ces éléments pour y effectuer différentes opérations telles que la modification des couleurs ou le rendu de la lumière.

Le pattern Visitor offre une solution pour séparer un algorithme de la structure d'objets sur laquelle il s'applique. L'intérêt de cette solution est la possibilité d'ajouter de nouvelles opérations à une structure d'objets existante, sans modifier cette structure. La Figure 2.3 présente un diagramme UML de cette solution.

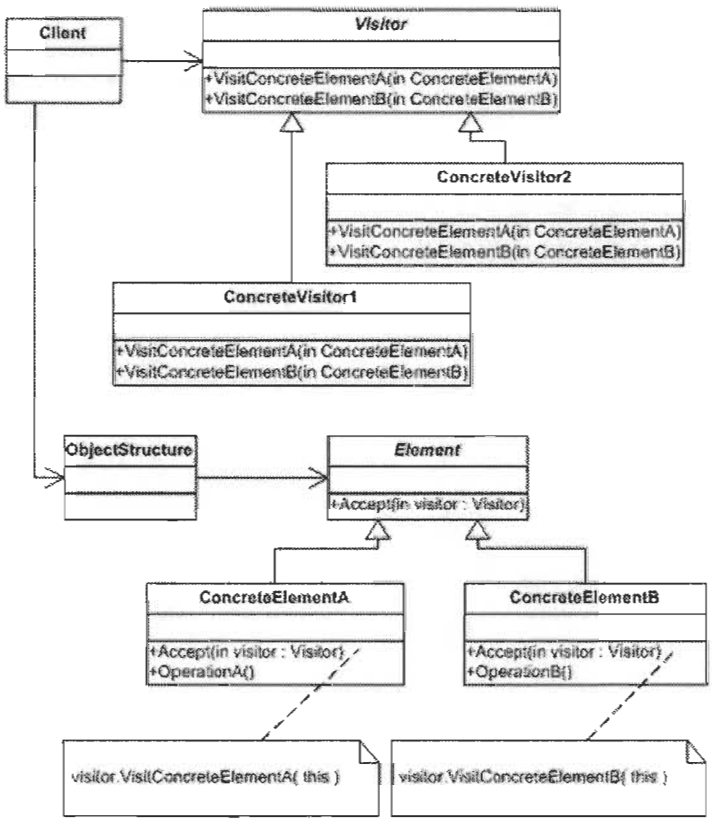

FIGURE 2.3 – Pattern visitor [Dofa 15].

2.2.4 Le pattern Prototype

Le pattern Prototype spécifie une interface qui permet de créer un clone de l'objet. Ce pattern est utilisé quand la création d'un objet est coûteuse (en temps ou ressources). Par exemple, un objet est créé à partir d'une base de données. La création de cet objet est très longue. Une fois cet objet créé, il est possible de le mettre en cache, puis d'obtenir des clones de cet objet sans avoir besoin d'en créer de nouveaux depuis la base de données. La Figure 2.4 présente un diagramme UML de cette solution.

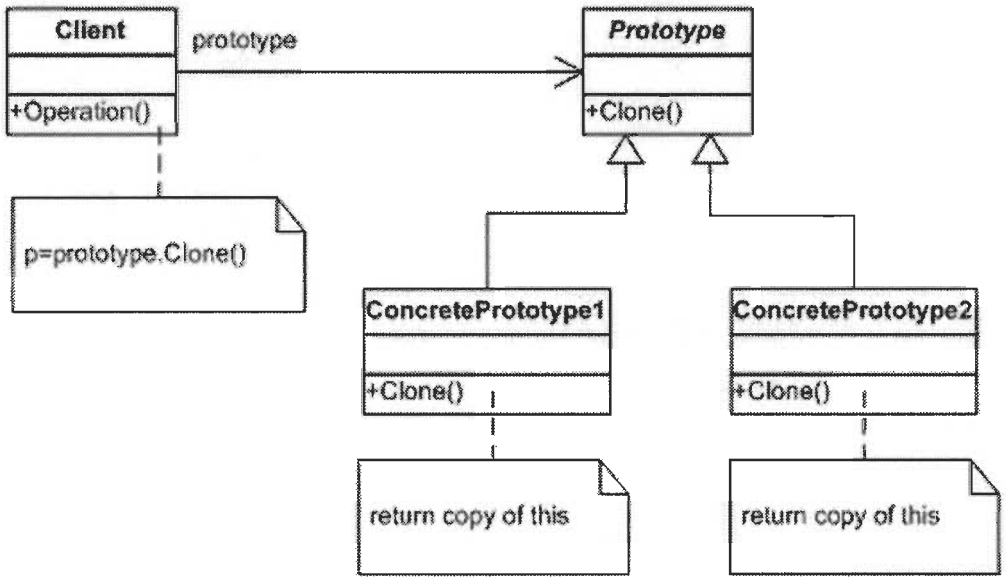

FIGURE 2.4 – Pattern prototype [Dofa 15].

Chapitre 3

La qualité logicielle

3.1 Introduction

La qualité logicielle est une appréciation d'un logiciel selon différents critères. Il existe de nombreux critères pour évaluer un logiciel, par exemple, la précision des résultats, la portabilité sur différentes plateformes ou encore la facilité à modifier le logiciel [ISO 01]. La qualité d'un logiciel dépend, entre autre, de la façon dont le logiciel a été développé et du processus utilisé pour le développer [Pres 05].

Pour rester compétitives, les entreprises de développement logiciel doivent livrer un produit de bonne qualité en respectant les délais et le budget. Cependant, la qualité des logiciels est souvent négligée lors du développement afin de respecter les délais et le budget [Kris 97]. Cette négligence est une erreur car les coûts des problèmes engendrés par un logiciel de mauvaise qualité sont supérieurs aux coûts nécessaires pour créer un logiciel de bonne qualité [Slau 98].

Ce sont les préoccupations liées à la qualité qui ont conduit dans les années 1960 au développement de la science de la création de logiciels, le génie logiciel.

La Section 3.2 propose une spécification de la qualité logicielle selon la norme ISO 9126. La Section 3.2.1 approfondit une des caractéristiques de la qualité logicielle, la maintenabilité. Cette dernière est approfondie car l'étude

présentée dans ce mémoire évalue la qualité logicielle au travers de deux perspectives particulières de la maintenabilité : la changeabilité et la testabilité. Finalement, la Section 3.3 décrit le principe des métriques, des outils fréquemment utilisés pour évaluer la qualité logicielle.

3.2 La norme ISO 9126

La norme ISO 9126 est un standard international pour évaluer la qualité logicielle [ISO 01]. Cette norme organise la qualité logicielle autour de six caractéristiques. Chacune de ces caractéristiques est divisée en sous caractéristiques, pour un total de 27 sous caractéristiques. L'intérêt principal de cette norme est de définir un vocabulaire commun pour parler des différents aspects de la qualité logicielle. Les six caractéristiques principales sont les suivantes :

La capacité fonctionnelle :	La capacité du logiciel à répondre aux besoins des usagers.
La facilité d'utilisation :	Caractérise l'effort nécessaire pour apprendre à manipuler le logiciel.
La fiabilité :	La capacité du logiciel à fournir des résultats corrects.
La performance :	Définit le rapport entre la qualité des résultats et les ressources qu'il est nécessaire d'utiliser pour les obtenir.
La maintenabilité :	Mesure de l'effort nécessaire pour corriger ou modifier le logiciel.
La portabilité :	L'aptitude d'un logiciel à fonctionner dans différents environnements matériels ou logiciels.

3.2.1 La maintenabilité

Le cycle de vie d'un logiciel commence généralement par une phase de développement qui est suivie par une phase d'exploitation du logiciel puis d'une phase de maintenance. Fréquemment, les équipes cherchent à optimiser la productivité de la phase de développement au détriment d'une future phase de maintenance. Ceci est une erreur car des études ont révélé que la phase de maintenance représente entre 50 et 70% du coût des logiciels [Somm 01] [Lee 00].

La maintenabilité est une des caractéristiques de la qualité logicielle définie par la norme ISO 9126. Cette caractéristique représente l'effort nécessaire pour corriger ou modifier un logiciel. Corriger ou modifier un logiciel est le coeur d'une activité de maintenance.

La norme ISO 9126 divise la maintenabilité en quatre sous caractéristiques :

- La facilité d'analyse.
- La stabilité.
- La changeabilité.
- La testabilité.

L'étude présentée dans ce mémoire considère la maintenabilité sous l'angle de la testabilité et de la changeabilité. C'est pourquoi les deux paragraphes suivants proposent une explication de la nature de ces deux sous caractéristiques.

La changeabilité

La norme [ISO 01] définit la changeabilité comme étant la facilité avec laquelle un logiciel peut être modifié. Ces modifications du logiciel peuvent

être effectuées dans le cadre de la correction d'erreurs, de l'ajout de nouvelles fonctionnalités ou de l'adaptation à un nouvel environnement. Ces modifications peuvent, par ailleurs, être effectuées dans le cadre d'opérations de remaniement. Donc, pour y effectuer un changement, un logiciel avec une bonne changeabilité demandera moins d'efforts qu'un logiciel avec une mauvaise changeabilité.

La testabilité

Selon la norme [ISO 01], la testabilité définit l'effort nécessaire pour valider un logiciel. En programmation orientée object, une part importante de ces validations sont effectuées avec des tests. Une part de la testabilité peut donc être déterminée par l'effort nécessaire pour réaliser ces tests.

3.3 Les métriques

Une métrique logicielle est une mesure d'un attribut d'un logiciel. Il existe une grande quantité de métriques différentes, des plus simples aux plus complexes. Au niveau du code source, des métriques simples peuvent être le nombre de paramètres d'une méthode, le nombre de méthodes d'une classe ou le nombre total de lignes de code d'un projet. La littérature scientifique propose aussi différentes métriques plus complexes obtenues en effectuant des opérations sur plusieurs métriques simples.

Les métriques sont largement utilisées en génie logiciel. Elles servent souvent d'indicateurs de la qualité logicielle. Pour illustrer de façon triviale l'utilisation d'une métrique, admettons qu'un logiciel de bonne qualité ne possède pas de classes de plus de 120 lignes de code. Le nombre de lignes de code d'une classe est une métrique. Généralement, les métriques sont nommées par un acronyme. Pour le nombre de lignes de code l'acronyme LOC ("Lines of code") est souvent utilisé. En utilisant cette métrique avec un outil approprié, il est possible d'analyser l'ensemble des classes d'un logiciel et de vérifier que la métrique LOC des classes est inférieure à 120.

En suivant le principe de l'exemple précédent, des études proposent des métriques qui sont représentatives des attributs de la qualité logicielle [Basi 96, Lore 94, Lanz 07, Bans 02]. Par exemple, [Sara 13] propose un catalogue de 570 métriques utilisables pour évaluer la maintenabilité. Il existe aussi des suites de métriques plus restreintes, par exemple [Chid 94] propose une suite de 6 métriques orientées objet couramment utilisées en génie logiciel.

Deuxième partie

État de l'art

Introduction de la Partie II

La norme [ISO 01] a posé une définition précise de la testabilité et de la changeabilité. Cependant, la manière d'évaluer ces deux caractéristiques n'est pas précisée. La communauté scientifique a donc effectué plusieurs travaux pour essayer de palier à ce manque [Ajrn 99, Ajrn 00, Sun 12, Kaba 01, Aris 06, Loza 08, Ayal 13, Chha 14, Flur 07, Heit 07, Rosk 14, Bind 94, Heit 07, Brun 06, McGr 96, Abdu 14, Abdu 15b, Abdu 15b]. Afin de saisir l'état actuel de la recherche sur ces sujets, les deux prochains chapitres (4 et 5) présentent une synthèse de ces travaux.

Avec des objectifs similaires à l'étude présentée dans ce mémoire, plusieurs études ont été effectuées pour évaluer l'impact de l'utilisation des design patterns. Comment les autres études ont procédé pour effectuer cette évaluation ? En présentant ces études, le chapitre 6 apporte la réponse à cette question.

Chapitre 4

Évaluation de la changeabilité

La changeabilité est une sous caractéristique importante de la maintenabilité. Cette affirmation est encore plus véridique dans les environnements où les logiciels sont amenés à être fréquemment modifiés [Ajrn 99].

Ce chapitre est une synthèse des travaux publiés dans la littérature qui définissent des métriques pour évaluer la changeabilité.

La plupart de ces études appartiennent à deux catégories. D'un coté, les études qui reposent sur l'introduction, par les auteurs de l'étude, de changements dans du code afin d'évaluer les effets de ces changements à l'aide de métriques[Ajrn 99, Ajrn 00, Sun 12, Kaba 01]. De l'autre, les études qui utilisent des dépôts logiciels pour analyser les changements qui ont été réalisés lors de projets réels [Aris 06, Loza 08, Ayal 13, Chha 14]. Il existe aussi quelques études qui n'appartiennent pas à ces deux catégories [Flur 07, Heit 07, Rosk 14].

Les sections 4.1, 4.2 et 4.3 présentent les études. La section 4.4 est une synthèse visuelle des relations entre les études.

4.1 Études qui introduisent des changements dans le code

L'étude présentée dans l'article *"A change impact model for changeability assessment in object-oriented software systems"* [Ajrn 99] porte sur un projet de 1044 classes réalisé en C++ par une entreprise de télécommunication. Dans cette étude, les classes du programme sont réparties en trois groupes. Cette répartition est effectuée selon la valeur de la métrique "Weighted Methods per Class (WMC)" [Chid 94] des classes. Une fois cette répartition terminée, des modifications sont effectuées sur les classes des trois groupes. Ensuite, à l'aide d'un modèle d'évaluation des changements, les conséquences engendrées par la modification des classes sont quantifiées. Avec ces données, l'étude démontre une relation entre l'importance des changements et la métrique WMC.

L'étude présentée dans l'article *"Design properties and object-oriented software changeability."* [Ajrn 00] considère 3 projets de tailles différentes (83, 584 et 1 226 classes) écrits en C++. Dans ces projets, des modifications (ajouts ; suppressions ; changements de type, de portée, de valeur de retour et de nom) sont effectuées au niveau des classes, des méthodes et des variables. Ensuite, à l'aide d'un modèle d'évaluation des changements, les répercutions de ces modifications sont mesurées. Ces mesures concernent 5 métriques issues de la suite de C&K (WMC, DIT, NOC, CBO, RFC) [Chid 94] et 4 métriques définies par les auteurs de l'article en apportant des modifications aux métriques proposées par C&K (métriques nommées : NOC, CBO_NA, CBO_IUB, CBO_U). En réalisant des études statistiques sur les données obtenues, les auteurs montrent que CBO_UIB a une corrélation très forte avec l'importance des changements. Dans plusieurs cas, selon le projet et le type de changements considérés, WMC, CBO, NOC et RFC peuvent avoir des corrélations significatives avec l'importance des changements. Également, une absence totale de corrélation est observée entre la profondeur dans l'arbre d'héritage (DIT) et l'importance des changements.

L'étude présentée dans l'article *"A change proposal driven approach for changeability assessment using fca-based impact analysis."* [Sun 12] considère trois systèmes différents de 26, 192 et 189 classes. Sur le même principe que les études précédentes [Ajrn 99] et [Ajrn 00], des modifications sont effectuées dans les systèmes. Ensuite, afin de mesurer l'impact des changements, un modèle d'évaluation du changement plus avancé que celui proposé par [Ajrn 00] est utilisé. Ce modèle propose une nouvelle métrique nommée "Impactness" qui permet d'évaluer la capacité du système à absorber un changement donné. Selon les auteurs de l'étude, "Impactness" permet de prédire la changeabilité des classes des systèmes.

L'étude présentée dans l'article *"Class cohesion as predictor of changeability : An empirical study."* [Kaba 01] considère les trois mêmes projets que l'étude [Ajrn 00], c'est à dire trois projets de tailles différentes (83, 584 et 1 226 classes) écrits en C++. L'objectif des auteurs de l'étude est de valider deux métriques de cohésion, LCC et LCOM, comme des indicateurs de changeabilité. Pour y parvenir, deux approches différentes sont proposées pour valider ces métriques.

La première approche se base sur la supposition qu'une classe fortement cohésive a un faible couplage et qu'une classe peu cohésive a un fort couplage. Des études statistiques sont donc effectuées pour évaluer la corrélation entre la cohésion (mesurée par les métriques LCC et LCOM) et le couplage (mesuré par les métriques CBO et RFC).

La seconde approche effectue des modifications dans la signature des classes. Avec un modèle d'évaluation des changements, les répercutions des modifications sont mesurées. Des études statistiques sont réalisées pour trouver des relations entre les métriques proposées (LCC et LCOM) et l'importance des changements.

La première approche ne montre pas de corrélation entre les mesures des supposées métriques de cohésion et les mesures des métriques relatives au

couplage. La seconde approche ne montre pas de corrélation entre les mesures des supposées métriques relatives à la cohésion et l'importance des changements. D'après cette étude, LCC et LCOM ne seraient donc pas de bons indicateurs de changeabilité.

4.2 Études qui exploitent les données des dépôts

L'étude présentée dans l'article *"Empirical assessment of the impact of structural properties on the changeability of object-oriented software."* [Aris 06] analyse les changements survenus sur un projet écrit par 3 développeurs en Java et C++. Le projet est suivi pendant 5 mois, 39 changements y sont relevés et 10 changements sont retenus pour l'étude. Au début des 5 mois du suivi, un procédé est mis en place pour collecter différentes informations lors du processus de développement. Ce procédé collecte des informations telles que le nombre d'heures nécessaires pour effectuer le changement ou les modifications du code engendrées par les changements. A la fin des 5 mois du suivi, des études statistiques sont réalisées pour trouver des relations entre plusieurs métriques relatives aux classes ou au système et les informations relatives aux changements collectés. Ces études statistiques font ressortir un modèle basé sur 2 métriques OMAEC_CP (basée sur l'accès aux attributs d'une classe par d'autres classes) et CS_CP (basée sur le nombre de lignes de code non commentées) qui permet d'expliquer 80% de la variance de l'effort du changement. Le détail du calcul des deux métriques est disponible dans l'article.

L'étude présentée dans l'article *"Assessing the effect of clones on changeability."* [Loza 08] porte sur l'évaluation de l'impact du code dupliqué (les clones) sur la changeabilité. Pour évaluer cet impact, à partir des données présentes dans des dépôts de code source (CVS), deux indicateurs de changeabilité sont calculés : "likelihood" et "impact of change" (la description du calcul de ces deux facteurs est disponible dans l'étude). Ensuite, une analyse

statistique entre ces deux indicateurs et les clones présents dans les dépôts confirme que le code dupliqué impacte la changeabilité.

L'étude présentée dans l'article *"An assessment of changeability of open source software."* [Ayal 13] analyse les logs de 62 tâches de maintenance sur le logiciel OpenBravoPOS entre la version 2.10 et la version 2.20. À partir des journaux, ils parviennent à extraire les changements qui ont été effectués et les classes impactées par ces changements. Avec ces informations, des études statistiques sont réalisées pour analyser les relations entre 4 métriques (CBO, Ca, Ce et WMC) et les informations relatives aux changements. Ces études statistiques démontrent que les 4 métriques considérées : CBO, Ca, Ce et WMC sont de bonnes indicatrices de changeabilité.

L'étude présentée dans l'article *"Prediction of changeability for object oriented classes and packages by mining change history."* [Chha 14] utilise des techniques de Data Mining pour explorer les journaux présents sur des dépôts de code source (SVN) d'une application de gestion des ressources humaines écrite en JAVA. À partir de ces données, en considérant des informations telles que le couplage ou la fréquence des changements des classes, les auteurs proposent une nouvelle métrique de changeabilité appelée le "changeability index" qui permet d'évaluer la changeabilité d'une classe ou d'un package.

4.3 Autres études

L'étude présentée dans l'article *"Assessing changeability by investigating the propagation of change types."* [Flur 07] propose un catalogue de critères de changeabilité. En utilisant ces critères, les auteurs proposent un modèle d'évaluation du changement permettant d'obtenir un classement de la changeabilité des classes.

L'étude présentée dans l'article *"A practical model for measuring maintainability."* [Heit 07] présente un axe d'évaluation de la maintenabilité et de ses 4 sous caractéristiques. Le principe est le suivant : pour une caractéristique de maintenabilité, il faut lui associer les propriétés du code source qui sont supposées avoir un impact sur la caractéristique. Par exemple, la changeabilité est influencée par la complexité. Puis, pour chaque propriété, les métriques qui permettent d'évaluer cette propriété sont mesurées. Dans cet article, les auteurs associent la changeabilité à deux propriétés : la complexité et la duplication du code.

L'étude présentée dans l'article *"Predicting the changeability of software product lines for business application."* [Rosk 14] porte sur une plate-forme composée de 9 applications, pour un total de 33 139 lignes de code Java. L'objectif de cette étude est d'évaluer la pertinence du "Maintenability Index" pour représenter la changeabilité. Pour y parvenir, les auteurs réalisent des études statistiques pour trouver des relations entre le Maintenability Index, le Platform Responsibility (une métrique de couplage créée par les auteurs) et plusieurs valeurs moyennes de métriques des composants (Complexité cyclomatique, DIT, WMC et différentes métriques de taille). Les résultats démontrent une corrélation entre la mesure du Maintenability Index et la mesure du Platform Responsibility avec les métriques proposées.

4.4 Relation entre les études

La figure 4.1 permet de visualiser les liens entre les différentes études présentées dans les sections précédentes. Une flèche entre deux articles indique que l'article à la fin de la flèche fait référence à l'article au début de la flèche.

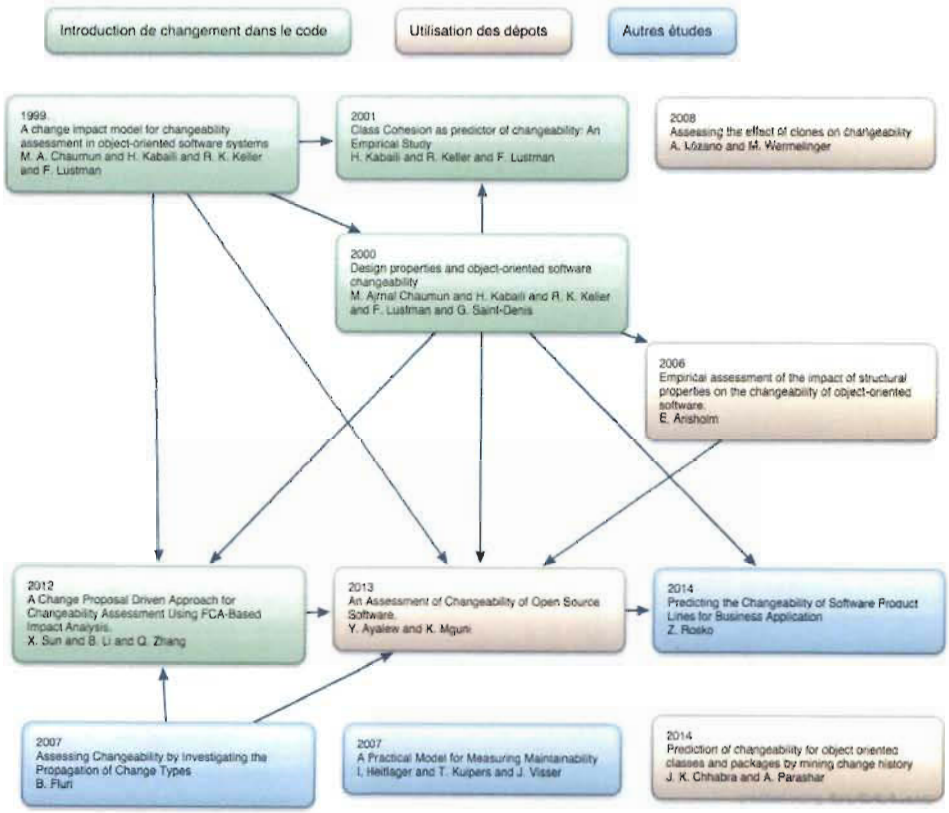

FIGURE 4.1 – Études sur la changeabilité.

4.5 Synthèse

Plusieurs auteurs développent leurs propres métriques pour évaluer la changeabilité : [Sun 12, Aris 06, Chha 14, Flur 07, Rosk 14]. Leurs approches sont souvent complexes et nécessitent des outils spécifiques pour pouvoir capturer ces métriques.

Aucune étude ne propose de résultats généralisables ou qui ont été reproduits dans d'autres études. L'hypothèse de l'étude [Rosk 14], qui assimile le Maintenability Index à la changeabilité, rend ces résultats inexploitables si une considération stricte de la changeabilité est étudiée. Une revue de la littérature globale de la maintenabilité proposerait beaucoup plus de résultats que ceux présentés ici.

Plusieurs études [Ajrn 99, Ajrn 00, Ayal 13] essayent de lier des métriques standardisées [Chid 94] avec la changeabilité. Le tableau 4.1 présente un condensé des résultats. Seules les études qui démontrent des corrélations sont répertoriées. Il est important de constater qu'uniquement 3 études lient la changeabilité avec des métriques standards, dont deux études du même auteur.

Métrique [1]	**Etudes**
WMC	[Ajrn 99, Ajrn 00, Ayal 13]
CBO	[Ajrn 00] [Ayal 13]
Ca	[Ayal 13]
Ce	[Ayal 13]
NOC	[Ajrn 00]
RFC	[Ajrn 00]

TABLE 4.1 – Métriques représentatives de la changeabilité.

1. La définition des acronymes de ces métriques est disponible page 50

Chapitre 5

Évaluation de la testabilité

La testabilité est une sous-caractéristique importante de la maintenabilité. Elle est même considérée par certains comme la sous-caractéristique la plus importante car les tests nécessitent une part considérable des ressources nécessaires pour développer et maintenir un logiciel [Chow 09].

Comment évaluer la testabilité d'un logiciel ? Les trois sections suivantes résument les études publiées dans la littérature sur ce sujet. Dans un premier temps, la section 5.1 présente les études qui abordent la testabilité sous un angle théorique. Ensuite, la section 5.2 présente les études qui essayent d'évaluer la testabilité avec des métriques. Finalement, la section 5.3 présente les études qui proposent des modèles afin d'évaluer la testabilité.

5.1 Études théoriques

Cette section présente des études théoriques sur l'évaluation de la testabilité.

L'étude présentée dans l'article *"Design for testability in object-oriented systems"* [Bind 94] présente sous forme de diagramme en arrêtes de poisson, différents facteurs qui influencent la testabilité du code. Cette étude propose également un regroupement de différentes métriques, telles que celles proposées par [Chid 94], pour mesurer la testabilité (table 5.1).

Complexité
Weighted method per class - Response for a class - Coupling between objects Nominal number of methods per class - Nominal number of function per class Class complexity - Procedure change state - Total number of methods per class Total number of function per class - Total number of procedure per class
Polymorphisme
Percent of non-overloaded call - Percent of dynamic calls Bounce-c - Bounce-s
Héritage
Number of root classes - Number of children - Depth of inheritance tree
Encapsulation
Lack of cohesion in methods - Percent public and protected - Percent access to data members

TABLE 5.1 – Métriques de testabilité proposées par [Bind 94].

L'étude présentée dans l'article *"A practical model for measuring maintainability"* [Heit 07] présente un axe d'évaluation de la maintenabilité au travers de ses 4 sous-caractéristiques(la facilité d'analyse, la stabilité, la changeabilité et la testabilité). Le principe de cet axe d'évaluation est le suivant : pour une sous-caractéristique de maintenabilité, il faut y associer les propriétés du code source qui sont supposées avoir un impact sur cette sous-caractéristique. Ensuite, chaque propriété doit être mesurée par des métriques. Par exemple, la testabilité (une sous-caractéristique) est influencée par la complexité (une propriété), puis il faudrait utiliser des métriques pour mesurer la complexité. Cette étude propose la taille, la complexité et les tests unitaires comme caractéristiques pour évaluer la testabilité. Cependant, elle ne donne pas les métriques nécessaires pour les mesurer.

5.2 Études empiriques

Cette section synthétise les résultats de plusieurs études [Brun 06] [Sing 10] et [Badr 11] qui utilisent une approche empirique pour évaluer la testabilité.

Ces études fonctionnent toutes sur un principe identique. En effet, leur objectif est de démontrer une relation entre différentes métriques relatives aux classes et différentes métriques relatives aux tests associés à ces classes. Par exemple, démontrer un lien entre le nombre de ligne de code des classes et des métriques relatives aux tests, telles que le nombre d'assertions, le nombre de lignes de code des tests ou le nombre de méthodes des tests.

En parvenant à démontrer un lien entre les métriques des classes et les métriques des tests qui y sont associés, il est alors possible de valider le fait que ces métriques relatives aux classes sont des indicateurs de leur testabilité. Ces liens sont démontrés grâce à des études statistiques effectuées sur différents projets. La première étude [Sing 10] porte sur Eclipse 3.0. La seconde étude [Brun 06] est une reprise de [Brun 04]. La différence tient dans le nombre de projets considérés pour l'étude empirique. Dans la plus récente [Brun 06], l'étude porte sur cinq projets écrits en Java : Apache Ant (172 906 lignes de code), DocGen (88 672 linges de code), Jackal (14 355 lignes de code), Monitor (10 269 lignes de code) et ZPC (9599 lignes de code). La troisième étude [Badr 11] reprend le contenu de l'article [Badr 10] en y ajoutant plusieurs analyses statistiques. Cette étude porte sur deux projets écrits en Java : Apache Ant (64 062 lignes de code) et JfreeChart (68 312 lignes de code).

La table 5.2 synthétise les résultats des trois études en présentant les métriques du code pour lesquelles une corrélation a été trouvée avec les métriques des tests.

Propriétés	Métriques de testabilité	[Sing 10]	[Brun 06]	[Badr 11]
Taille	Lines of code per class	X	X	X
	Number of methods	X	X	X
	Number of attributs	X		X
	Weighted methods per class	X	X	
Cohésion	Lack of cohesion of methods	X	X	X
	Lack of cohesion of methods *			X
	Lack of cohesion			X
	Information based cohesion	X		
	Tight Class Cohesion	X		
Couplage	Coupling between objects	X		
	Data abstraction coupling	X		
	Message passing coupling	X		
	Response for class	X	X	
	Fan out		X	
Héritage	Depth of inheritance tree	X	-	
	Number of children	X	-	
Polymorphisme	Number of methods overridden	X		

TABLE 5.2 – Métriques de testabilité validées par les études.

5.3 Études qui proposent des modèles

Les études suivantes n'évaluent pas directement la testabilité avec des métriques standards. Elles se placent à un plus haut niveau, en construisant un modèle à partir de plusieurs métriques. Les modèles sont souvent complexes et/ou peu détaillés dans les articles.

L'étude présentée dans l'article *"A measure of testing effort"* [McGr 96] propose un modèle d'évaluation de la testabilité au niveau des méthodes des classes. Ce modèle est basé sur une métrique de visibilité des composants des

méthodes : "Visibility Component". La métrique "Visibility Component" caractérise l'accessibilité des objets qui ont changé d'état lors de l'appel d'une méthode. L'avantage de ce modèle est de pouvoir être appliqué, si les spécifications sont suffisantes, dès la phase de design de l'application. Cette étude étant antérieure à la norme ISO 9126, la testabilité est évaluée en tant que capacité des tests à découvrir une faute. En appliquant leur modèle à plusieurs classes écrites en C++, les auteurs valident la pertinence du modèle pour capturer la testabilité dans les premières phases du développement. Cependant, la précision baisse quand le code est implémenté.

L'étude présentée dans l'article *"Modifiability : A key factor to testability"* [Abdu 14] établit une liste des facteurs qui influent sur la testabilité et propose un modèle d'évaluation de l'un d'entre eux, la "modifiability". A partir des travaux publiés dans la littérature, les facteurs qui influencent la testabilité sont liés à des caractéristiques du code, par exemple la "modifiability" est influencée par l'encapsulation, l'héritage et le couplage. Finalement, chaque caractéristique du code est reliée à des métriques couramment utilisées pour mesurer ces caractéristiques. A partir des caractéristiques du code mesurées par les métriques relatives aux nombres de méthodes des classes, à la profondeur maximum dans l'arbre d'héritage et au nombre d'associations des classes, les auteurs proposent un modèle d'évaluation de la "modifiability". Les auteurs valident le modèle en effectuant une étude empirique dans laquelle le modèle à été appliqué sur 28 diagrammes de classes.

L'étude présentée dans l'article *"Flexibility : A key factor to testability"* [Abdu 15b] propose un modèle d'évaluation d'un facteur qui influence la testabilité : la "flexibility". A partir des travaux publiés dans la littérature, les facteurs qui influencent la "flexibility" sont liés à des caractéristiques du code, par exemple la "flexibility" est influencée par l'encapsulation, la cohésion, l'héritage et le couplage. Ensuite, chaque caractéristique du code est reliée à des métriques couramment utilisées pour mesurer ces caractéristiques. À partir des caractéristiques du code mesurées par les métriques

"Data Access Metrics", "Direct Class Coupling", "Cohesion Among Methods of Class" et "Measure of Functional Abstraction", les auteurs proposent un modèle d'évaluation de la "flexibility". En appliquant le modèle sur les données traitées dans cet article [Bans 02], des études statistiques sont utilisées pour valider le modèle.

L'étude présentée dans l'article *"Testability Measurement Model for Object Oriented Design (TMMOOD)"* [Abdu 15a] est une agrégation des deux articles précédemment résumés, [Abdu 15b] et [Abdu 14]. A partir des modèles d'évaluation de la "flexibility" et de la "modifiability", un modèle global d'évaluation de la testabilité est proposé. Ensuite, ce modèle est testé et validé à travers une étude empirique sur les données extraites de l'étude [Bans 02].

5.4 Relation entre les études

La figure 5.1 représente l'ensemble des articles présentés dans les sections précédentes. Une flèche entre deux articles indique que l'article à la fin de la flèche fait référence à l'article au début de la flèche. "Études générales" est à comprendre dans le sens étude théorique.

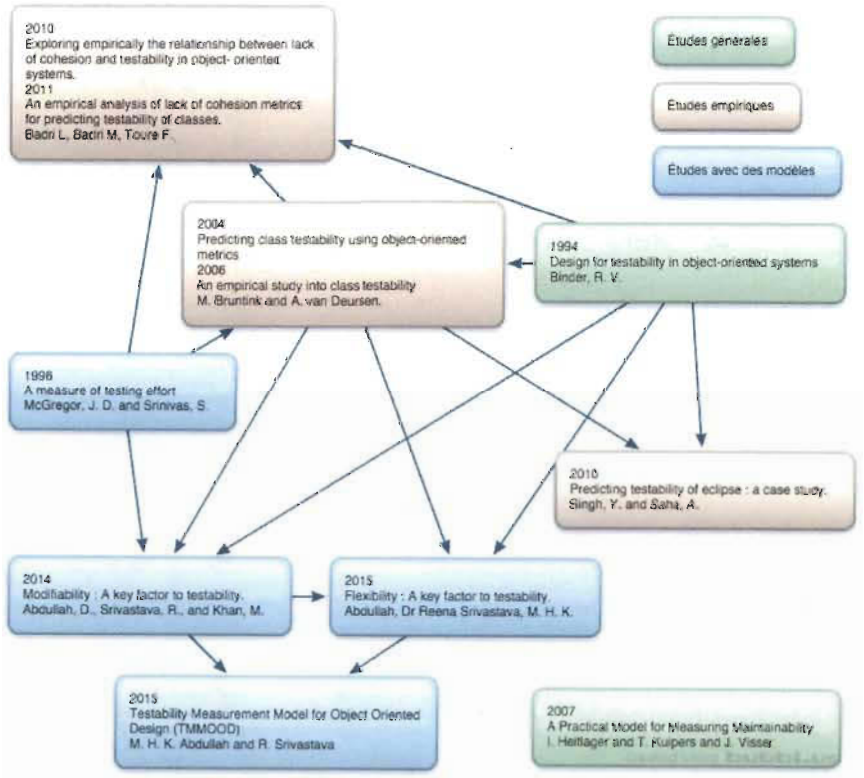

FIGURE 5.1 - Études sur la testabilité.

Chapitre 6

Comment évaluer l'impact des design patterns ?

De nombreuses études se sont penchées sur l'évaluation des impacts liés à l'utilisation des design patterns [Beck 96, Hust 01, Prec 01, Voka 04a, Amou 06, Aver 07, Khom 08, Alsh 11, Ampa 12, Hege 12]. Selon le cadre d'évaluation et la méthodologie utilisée, les résultats de ces études sont constrastés.

Cependant, ce chapitre ne traite pas des résultats que ces études ont permis d'obtenir, mais plutôt des différentes solutions qui ont été utilisées afin de réussir à obtenir ces résultats. En effet, la principale difficulté pour évaluer l'impact des design patterns est liée à la méthodologie utilisée. C'est pour cette raison que les auteurs des différentes études utilisent des approches diversifiées pour répondre à la question : comment évaluer l'impact des design patterns ?

Les analyses effectuées dans les études portant sur l'évaluation de l'impact des design patterns peuvent être soit qualitatives soit quantitatives. Étant donné que les travaux présentés dans ce mémoire utilisent une approche quantitative, la section 6.1 regroupe l'ensemble des études quantitatives publiées qui évaluent l'impact des design patterns. Afin d'obtenir un aperçu de méthodes utilisables dans des études qualitatives, celles-ci sont évoquées dans la section 6.2 sous forme de deux exemples.

6.1 Études quantitatives

Les outils actuels permettent d'effectuer les mesures nécessaires aux études quantitatives relativement facilement. La principale difficulté est d'obtenir un volume de données (code source ou modèle) qui permet d'obtenir des résultats significatifs et interprétables .

Les études qualitatives pour mesurer l'impact des design patterns peuvent être divisées en deux catégories. Dans la première catégorie, section 6.1.1, les design patterns sont introduits pour l'étude. Dans la seconde catégorie, section 6.1.2, des projets existants sont analysés pour y détecter les design patterns.

6.1.1 Introduction des design patterns

Ces approches ont pour objectif d'obtenir au moins deux versions différentes d'un design : une version avec le design pattern et une (des) solution(s) alternative(s). Grâce à ces différentes versions, il est possible d'effectuer des mesures comparatives.

Approches basées sur des modèles théoriques

Dans [Hust 01], les auteurs considèrent que pour chaque design pattern, il existe une solution correspondante sans le design pattern dite "a non-pattern solution". Ils se basent alors sur les modèles de ces deux versions pour effectuer leurs mesures.

Dans [Ampa 12], les auteurs recherchent dans la littérature, dans les projets open-source et dans leurs expériences personnelles, les solutions alternatives possibles à un design pattern. À partir des modèles de ces solutions (le design pattern et les alternatives), ils identifient tous les changements majeurs qui pourraient survenir dans la pratique, par exemple l'ajout de nouvelles classes ou de méthodes. Puis, ils appliquent ces changements aux modèles identifiés précédemment. Ils ont alors à leur disposition plusieurs modèles

qui représentent de façon abstraite les différents états possibles des designs. Ils peuvent y effectuer des mesures et comparer le design pattern avec la(les) solution(s) alternative(s).

Approche basée sur du code

Dans [Alsh 11], les auteurs prennent trois petits systèmes open source (entre 300 et 12 000 lignes de code). En se basant sur leurs connaissances personnelles, ils appliquent manuellement des design patterns sur les projets. Ils obtiennent alors une version avant et une version après refactoring sur lesquelles ils peuvent effectuer des mesures comparatives.

Approches mixtes code et modèles

Dans [Amou 06], les auteurs commencent par extraire à partir de projets open source leur représentation UML. Puis, sur ces modèles, ils appliquent un algorithme génétique pour détecter des opportunités intéressantes d'applications des design patterns. Ils obtiennent alors plusieurs versions des projets initiaux sur lesquels des design patterns ont été appliqués.

Approches avec des participants

Dans cette étude [Prec 01] et de manière plus approfondie dans celle-ci [Voka 04b], les auteurs demandent à une équipe d'ingénieurs d'appliquer des modifications sur des projets avec et sans design patterns. Ils comparent les performances des participants à effectuer les modifications.

6.1.2 Analyse de code existant

Certaines études [Hege 12, Voka 04a, Aver 07] évaluent l'impact que les design patterns ont eu dans des projets existants. Les études prennent en considération plusieurs versions d'un projet et essayent de trouver des corrélations entre les design patterns et l'évolution des différents attributs du code.

Dans ces approches, la difficulté réside dans la détection des design patterns. De nombreuses techniques comme celles présentées dans [Gueh 09, Tsan 06, Anto 98a, Anto 98b, Tahv 03, Fere 05, Kell 99] ont été mises au point pour répondre à ce problème. Cette liste n'est pas exhaustive et une étude plus approfondie pourrait être faite sur les outils et les techniques de patterns-mining.

6.2 Études qualitatives

Ce type d'études fait appel à des experts qui donnent leurs avis sur les design patterns.

Dans [Khom 08], un questionnaire est envoyé à des développeurs expérimentés par e-mail. Ce questionnaire demande d'évaluer de manière subjective l'impact du design pattern sur différents critères de qualité. L'évaluation porte sur 5 niveaux allant de très négatif à très positif. L'étude a duré 4 mois. Les réponses de 20 ingénieurs avec une expérience suffisante ont été retenues.

Dans [Beck 96], l'évaluation est faite par des utilisateurs expérimentés et reconnus dans le domaine de l'industrie. Ils donnent leur retour d'expériences sur l'utilisation des design patterns.

6.3 Conclusion

Les manières d'aborder l'étude de l'impact des design patterns sont variées.

Les études quantitatives basées sur la génération de design patterns sont les plus nombreuses. D'un point de vue technique, elles sont les plus simples à mettre en place. Parmi celles-ci, les études dans lesquelles les auteurs créent eux mêmes les design patterns [Hust 01, Ampa 12, Alsh 11, Amou 06] souffrent d'un biais important car la façon dont les auteurs utilisent les design patterns peut entrainer d'importantes variations dans les résultats. Pour palier à ce biais, des études [Prec 01, Voka 04b] font appel à de nombreux participants pour obtenir des données plus significatives. Ces dernières approches sont intéressantes mais nécessitent des moyens importants.

Les études quantitatives basées sur l'analyse du code existant [Hege 12, Voka 04a, Aver 07] sont intéressantes car elles permettent d'obtenir des résultats à partir de cas où les design patterns ont été utilisés dans un développement réel. Cependant, ces études sont peu nombreuses car la détection de design patterns dans du code est complexe et nécessite des outils avancés de pattern mining.

Troisième partie

Étude expérimentale

Chapitre 7

Méthodologie de l'étude

Dans la partie I, a été abordée la notion de qualité logicielle et a été introduit le concept de design pattern. Par la suite, dans la partie II ont été présentées différentes études traitant de l'évaluation de la qualité sous deux angles spécifiques : la testabilité et la changeabilité. Également, la partie II a fait référence à plusieurs études qui portent sur l'évaluation de l'impact des design patterns. Ces deux parties sont une mise en contexte de la méthodologie présentée dans ce qui suit.

Ce chapitre est divisé en 5 sections. Les sections 7.1, 7.2 et 7.3, sous la forme de l'approche GQM, définissent les éléments d'évaluation utilisés dans cette étude. La section 7.4 détaille le processus de collecte des données. Finalement, la section 7.5 décrit les outils utilisés pour réaliser l'étude.

Dans les trois sections suivantes, nous allons utiliser l'approche GQM (Goal Question Metric) [Cald 94] pour définir un ensemble de métriques pertinentes pour cette étude. Cette approche est divisée en trois étapes. La première étape est la définition des objectifs. La seconde étape pose une série de questions qui permettent d'évaluer si ces objectifs sont atteints. La troisième et dernière étape est une définition des métriques qui permettent de répondre à ces questions.

7.1 Objectifs

Le point de départ de la méthode GQM [Cald 94] consiste à définir les objectifs. Les objectifs de cette étude portent sur l'évaluation de l'impact

des design patterns sur la testabilité et la changeabilité.

Divisée en deux objectifs, nous obtenons :

Objectif 1 : Évaluer l'impact des design patterns sur la testabilité.

Objectif 2 : Évaluer l'impact des design patterns sur la changeabilité.

7.2 Questions

Posons les questions qui permettront d'affiner le périmètre de l'étude et de construire une réflexion pertinente. Ces questions déboucheront sur la définition des métriques dans la section 7.3.

Question 1

Afin d'étudier l'impact des design patterns sur la testabilité et la changeabilité, concentrons nous sur un événement particulier, à savoir, l'introduction d'un design pattern dans une classe existante. Il y a introduction lorsque les conditions suivantes sont respectées :

- La classe est présente dans une version N d'un logiciel sans être impliquée dans un design pattern.
- La classe est présente dans la version suivante N+1 du logiciel et est impliquée dans un design pattern.

Dans ce cas, existe-t'il une différence en terme de testabilité ou de changeabilité entre la classe avant l'introduction du design pattern et la classe après l'introduction du design pattern ? Divisée en deux questions nous obtenons :

Question 1.1 : Est-ce que l'introduction d'un design pattern a un impact sur la testabilité d'une classe ?

Question 1.2 : Est-ce que l'introduction d'un design pattern a un impact sur la changeabilité d'une classe ?

Question 2

Pour une version donnée d'un logiciel, comparons l'ensemble des classes qui sont impliquées dans un design pattern avec l'ensemble des classes qui ne sont pas impliquées dans un design pattern. Étudions si les classes impliquées dans un design pattern ont une testabilité et une changeabilité différentes des classes qui ne sont pas impliquées dans un design pattern.

Question 2.1 : Pour l'ensemble des classes d'une version, est-ce que les classes impliquées dans un design pattern ont une testabilité différente des classes qui ne sont pas impliquées dans un design pattern ?

Question 2.2 : Pour l'ensemble des classes d'une version, est-ce que les classes impliquées dans un design pattern ont une changeabilité différente des classes qui ne sont pas impliquées dans un design pattern ?

Questions subsidiaires

Apporter des réponses aux questions 1 et 2 permettrait d'obtenir des éléments afin d'atteindre les objectifs 1 et 2. Cependant, ces questions soulèvent plusieurs questions subsidiaires.

Question subsidiaire 1 : Comment évaluer la testabilité des classes ?

Question subsidiaire 2 : Comment évaluer la changeabilité des classes ?

La réponse a ces deux premières questions passe par l'utilisation de métriques de classes, représentatives de la testabilité ou de la changeabilité. Le choix des métriques est développé dans la section 7.3.

Question subsidiaire 3 : Comment définir l'implication des classes dans des design patterns ?

Une classe est impliquée dans un design pattern si elle fait partie d'un des composants du design pattern. Une classe peut être impliquée dans zéro, un ou plusieurs patterns différents. Donc, évaluer l'implication d'une classe dans des design patterns revient à compter le nombre de design patterns différents dans lesquels la classe est impliquée. Plus de détails sur l'outil pour détecter les design patterns sont disponibles dans la sous section 7.5.1.

Questions 1'

Les notions apportées par les questions subsidiaires permettent de revenir sur les questions 1.1 et 1.2 afin de les reformuler d'une façon plus précise :

Question 1.1' : Est-ce que l'introduction d'un design pattern sur une classe provoque un changement significatif des valeurs des métriques de testabilité ?

Question 1.2' : Est-ce que l'introduction d'un design pattern sur une classe provoque un changement significatif des valeurs des métriques de changeabilité ?

Pour répondre à ces questions, il va être nécessaire d'effectuer une analyse statistique sur la valeur des métriques de changeabilité et de testabilité. L'analyse comparera les valeurs des métriques avant l'introduction d'un design pattern avec les valeurs des métriques après l'introduction d'un design pattern. Le processus de collecte de ces données est détaillé dans la section 7.4.

Questions 2'

Les notions apportées par les questions subsidiaires permettent également de revenir sur les questions 2.1 et 2.2 afin de les reformuler d'une façon plus précise :

Question 2.1' : Pour l'ensemble des classes d'une version, est-ce que la valeur des métriques de testabilité des classes est corrélée au nombre de design patterns dans lequel elles sont impliquées ?

Question 2.2' : Pour l'ensemble des classes d'une version, est-ce que la valeur des métriques de changeabilité des classes est corrélée au nombre de design patterns dans lequel elles sont impliquées ?

Pour répondre à ces questions, il va être nécessaire d'effectuer une analyse des corrélations entre le nombre de design patterns dans lequel les classes sont impliquées et les différentes métriques de changeabilité et de testabilité. Le processus de collecte de ces données est détaillé dans la section 7.4.

7.3 Métriques

7.3.1 Nombre de design patterns

Les questions 1' et 2' mettent en évidence la première métrique nécessaire, à savoir le nombre de design patterns dans lequel une classe est impliquée.

Étant donné qu'il existe une grande quantité de design patterns différents, quels sont les patterns à comptabiliser pour déterminer le nombre de design patterns dans lequel une classe est impliquée ?

L'outil Ptidej, présenté dans la sous section 7.5.1, permet de détecter plusieurs patterns du GOF. D'après l'étude [Gueh 08], les patterns "Visitor", "Observer", "Composite" et "Prototype" sont les 4 design patterns détectés par l'outil avec les meilleurs taux de précision. De plus, ils permettent de représenter les 3 catégories différentes de pattern GOF (comportementaux, structuraux et de construction). C'est pour ces raisons que les occurrences de ces 4 design patterns seront celles considérées pour cette étude.

7.3.2 Changeabilité et testabilité

Les questions 1' et 2' appellent aussi la définition de métriques pour définir des cadres d'évaluation de la testabilité et de la changeabilité.

Deux facteurs limitent le choix des métriques disponibles pour définir les cadres d'évaluation.

Premier facteur limitant : L'étude porte sur l'ensemble des versions disponibles de plusieurs systèmes écrits en Java (section 7.6). Afin de répondre aux questions 1' et 2', 123 955 classes réparties dans 161 versions de 5 logiciels différents sont étudiées. Ce volume d'informations important nécessite une automatisation des processus de collecte des données. Les outils de collecte des métriques qui permettent de traiter un tel volume de données ne sont pas nombreux. Pas nombreux est même un euphémisme, car un seul outil fonctionnel a été trouvé. Les possibilités offertes par cet outil limitent donc le choix des métriques utilisables. De plus amples informations sur cet outil sont disponibles dans la section 7.5.1.

Second facteur limitant : Selon la revue de la littérature effectuée dans le chapitre 4, la rareté des études pertinentes sur la changeabilité limite fortement le choix des métriques utilisables pour évaluer la changeabilité.

En s'appuyant sur les revues de la littérature effectuées dans les chapitres 4 et 5 , neuf métriques (DIT, NOC, CE, CA, CBO, RFC, WMC, LOC, LCOM, TMM) ont été sélectionnées pour évaluer la testabilité et six métriques (NOC, RFC, CBO, WMC, CA, CE) ont été sélectionnées pour évaluer la changeabilité.

Les métriques sélectionnées sont regroupées dans les figures 7.2 et 7.1. Ces figures contiennent trois colonnes. Dans la colonne du centre, se trouvent les métriques relatives aux deux cadres d'évaluation. Dans la colonne de gauche, sont listées les études qui proposent les métriques qui sont pointées par les

flèches. Dans la colonne de droite, sont présents les attributs du code évalués par les différentes métriques.

Parmi ces neuf métriques retenues, huit sont des métriques issues de la suite de C&K [Chid 94]. La dernière, TMM, est un modèle de testabilité proposé par [Abdu 15a].

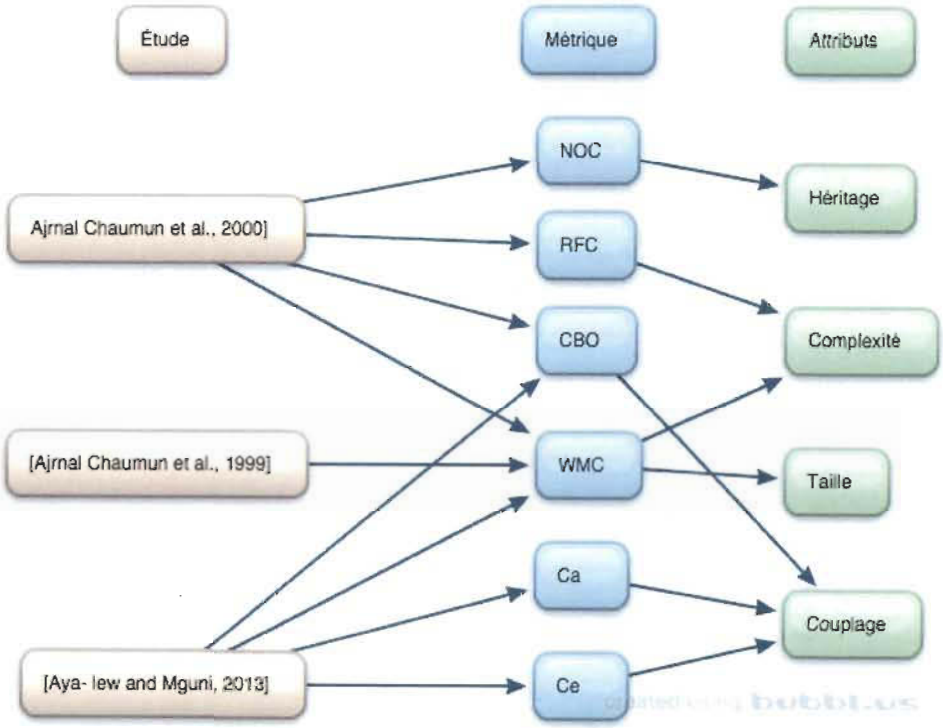

FIGURE 7.1 – Cadre d'évaluation de la changeabilité.

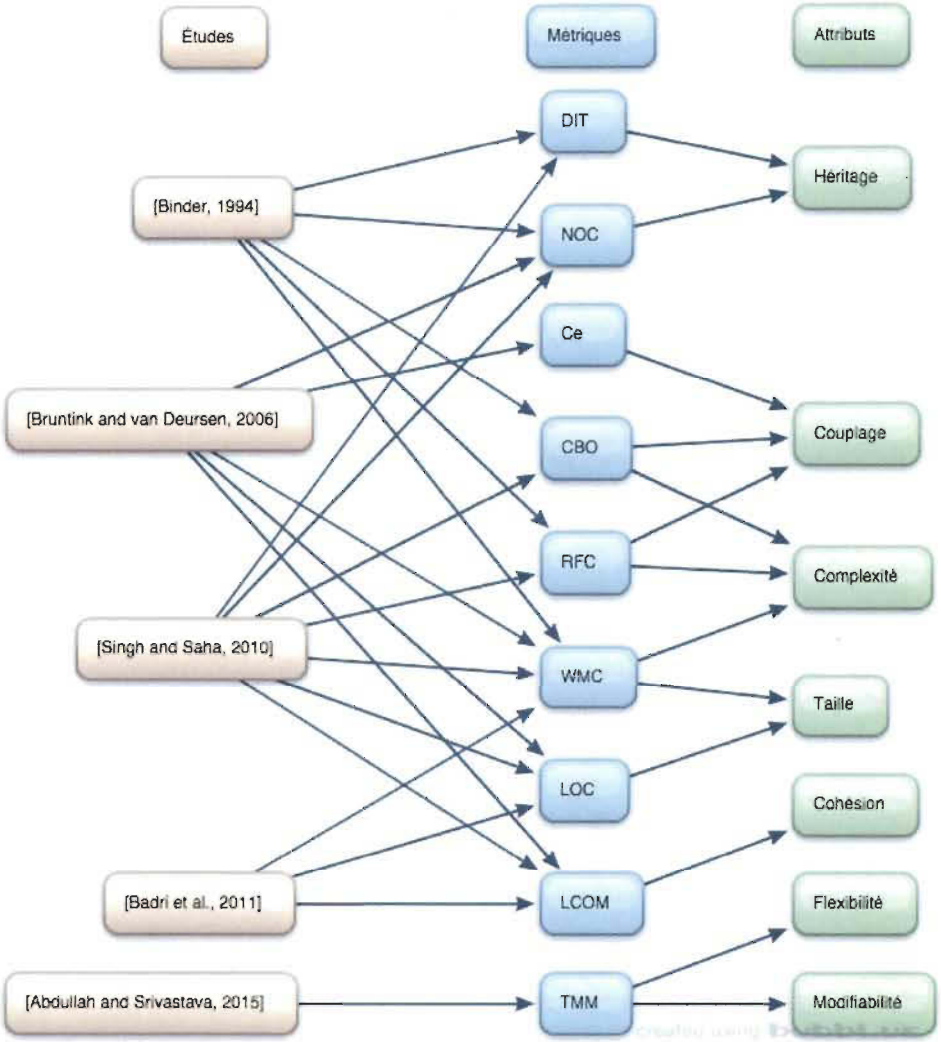

FIGURE 7.2 – Cadre d'évaluation de la testabilité.

Acronymes des métriques [1]

WMC "Weighted methods per class" définit le nombre de méthodes dans une classe.

RFC "Response for a Class" définit le nombre de méthodes différentes qui peuvent être exécutées quand une méthode d'une classe est appelée.

CBO "Coupling between object classes" définit le nombre de classes couplées avec une classe. Ce couplage peut être dû à un appel de méthode, à un accès à un attribut, à l'héritage, aux arguments passés dans une méthode, au type de retour d'une méthode ou aux exceptions.

Ca "Afferent couplings" définit le nombre de classes qui utilisent une classe.

Ce "Efferent couplings" définit le nombre de classes utilisées par une classe.

DIT "Depth of Inheritance Tree" définit la profondeur entre une classe et le haut de l'arbre d'héritage (la classe "Object" en Java).

NOC "Number of Children" définit le nombre de classes héritant d'une classe.

LOC "Lines of Code" définit le nombre de lignes de code de la classe (sans les lignes vides et les commentaires).

LCOM "Lack of cohesion in methods" définit le nombre de méthodes d'une classe qui ne partagent pas un attribut de cette classe.

TMM "Testability Measurement Model" est un modèle d'évaluation de la testabilité [Abdu 15a].

1. La définition de ces métriques est une traduction libre des definitions disponibles sur le site `http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/metric.html` relatif à l'outil CKJM Extended [Jure 10]

7.4 Processus de collecte des données

Cette section détaille les processus mis en place pour collecter les données utilisées pour répondre aux questions 1' et 2'.

7.4.1 Collecte pour les Questions 1.1' et 1.2'

Dans un premier temps, il faut détecter des classes, qui sont présentes dans deux versions successives d'un logiciel et qui, entre ces deux versions, ont subi une modification afin de les intégrer dans un design pattern. Autrement dit, une classe qui est présente dans la version 'v' ainsi que dans la version suivante 'v+1'. Dans la version 'v', la classe n'est pas impliquée dans un design pattern, dans la version 'v+1', la classe est impliquée dans un design pattern.

Le logiciel Ptidej [Gueh 08] présenté dans la section 7.5.1 permet de détecter les occurrences des différents design patterns dans du code Java. Ce logiciel a été utilisé afin de détecter, pour une version d'un logiciel, toutes les classes impliquées dans un design pattern. En comparant les résultats fournis par Ptidej entre une version "v" et une version "v+1", il est possible d'extraire les classes qui ont subi l'introduction d'un design pattern.

Dans un second temps, suite à la détection des classes nouvellement impliquées dans un design pattern, il faut collecter les valeurs des métriques des classes avant et après leur implication dans le design pattern.

Cette collecte est réalisée à l'aide de l'API CKJM [Jure 10], celle-ci permet de calculer la valeur des métriques de testabilité et de changeabilité pour l'ensemble des classes d'une version.

Comme le nombre d'introductions d'un design pattern sur une classe entre deux versions est faible (au maximum quelques occurrences), un nombre

important de versions ont été analysées afin d'obtenir un nombre statistiquement significatif de cas d'introduction. En se basant sur Ptidej et sur l'API CKJM, nous avons développé un outil pour automatiser cette collecte (présenté dans la Section 7.5.2). Cet outil permet, parmi d'autres fonctionnalités, d'extraire tous les cas d'introduction d'un design pattern pour l'ensemble des versions d'un logiciel, puis de collecter les métriques relatives aux classes impliquées dans ces introductions. Grâce à cet outil, nous obtenons donc une liste de l'ensemble des cas d'introduction d'un design pattern sur une classe, avec pour chaque cas, les valeurs des métriques de changeabilité et de testabilité avant et après l'introduction.

À partir de cette liste de résultats, un test statistique est effectué entre les valeurs des métriques avant et après l'introduction pour détecter si cette introduction produit un changement significatif sur la testabilité et la changeabilité. Ces résultats sont présentés dans le chapitre 8.

7.4.2 Collecte pour les Questions 2.1' et 2.2'

Pour apporter des éléments de réponse aux Questions 2, il faut collecter deux types de données différentes. Dans un premier temps, il faut obtenir le nombre de design patterns dans lequel les classes sont impliquées. Puis dans un second temps, il faut obtenir les valeurs des métriques de changeabilité et de testabilité de ces classes.

Comme dans la sous-section précédente relative aux Questions 1, Ptidej est utilisé pour détecter les classes impliquées dans un design pattern et CKJM est utilisé pour collecter les métriques. Ces données sont agrégées et analysées grace à l'outil que nous avons développé (outil présenté dans la Section 7.5.2).

À partir des données relatives à toutes les classes d'une version, une étude de corrélation est effectuée pour déterminer s'il existe un lien entre le nombre de design patterns dans lequel les classes sont impliquées et les métriques de testabilité et de changeabilité.

7.5 Outils

Trois logiciels différents ont permis la collecte et l'analyse des données.

Le premier logiciel, nommé Ptidej [Gueh 08], permet de détecter des design patterns dans du code source Java ou des fichiers binaires. C'est grâce à ce logiciel que l'implication des classes dans des design patterns est obtenue.

Le second logiciel, CKJM Extended [Jure 10], permet de calculer un ensemble de métriques issues de la suite de C&K [Chid 94].

Le dernier logiciel, Commons Math [Comm 16], est une API Java développée par Apache qui permet, entre autres, de supporter des calculs mathématiques, d'effectuer des analyses de corrélation (Pearson) et des tests statistiques.

Etant donné que ces logiciels fournissent en sortie des données dans des formats incompatibles entre eux et qu'ils ne permettent de traiter qu'une version d'un logiciel à la fois, il a été nécessaire d'intégrer ces trois logiciels dans un logiciel de plus haut niveau afin d'obtenir les résultats souhaités.

La sous section 7.5.1 présente succinctement les trois logiciels tiers utilisés, puis la sous section 7.5.2 traite du logiciel créé pour cette étude.

7.5.1 Logiciels tiers

PtiDej

Ptidej [Gueh 08] est un logiciel créé à l'École des Mines de Nantes ainsi qu'à l'École Polytechnique de Montréal par Yann-Gaël Guéhéneuc et le Ptidej Team. Il peut être utilisé via une interface graphique, en lignes de commandes ou dans Eclipse. En prenant en entrée un répertoire avec des classes Java ou du bytecode Java, le logiciel Ptidej utilise un procédé de reverse engineering pour créer un modèle du logiciel. À partir de ce modèle, avec un

système de contraintes, le logiciel détecte des structures susceptibles d'être des occurences des design patterns. 14 patterns du GOF sont détectables. La figure 7.3 montre un exemple de données obtenues avec le logiciel Ptidej.

```
Micro-architecture 92 similar to Composite Design Motif with a confidence of 42
                component = org.jfree.chart.ChartTheme
                composite = org.jfree.chart.ChartFactory
                leaf = org.jfree.chart.StandardChartTheme
                Name = Composite Design Motif
                Sign = Positive
                XCommand =

Micro-architecture 78 similar to Composite Design Motif with a confidence of 42
                component = org.jfree.chart.annotations.CategoryAnnotation
                composite = org.jfree.chart.plot.CategoryPlot
                leaf-1 = org.jfree.chart.annotations.CategoryPointerAnnotation
                leaf-2 = org.jfree.chart.annotations.CategoryLineAnnotation
                leaf-3 = org.jfree.chart.annotations.CategoryTextAnnotation
                Name = Composite Design Motif
                Sign = Positive
                XCommand =

Micro-architecture 47 similar to Composite Design Motif with a confidence of 42
                component = org.jfree.chart.annotations.XYAnnotation
                composite = org.jfree.chart.plot.XYPlot
                leaf-1 = org.jfree.chart.annotations.XYTitleAnnotation
                leaf-10 = org.jfree.chart.annotations.XYPointerAnnotation
                leaf-11 = org.jfree.chart.annotations.XYShapeAnnotation
                leaf-2 = org.jfree.chart.annotations.XYImageAnnotation
                leaf-3 = org.jfree.chart.annotations.XYDataImageAnnotation
                leaf-4 = org.jfree.chart.annotations.XYBoxAnnotation
                leaf-5 = org.jfree.chart.annotations.XYDrawableAnnotation
                leaf-6 = org.jfree.chart.annotations.XYLineAnnotation
                leaf-7 = org.jfree.chart.annotations.XYTextAnnotation
                leaf-8 = org.jfree.chart.annotations.AbstractXYAnnotation
                leaf-9 = org.jfree.chart.annotations.XYPolygonAnnotation
                Name = Composite Design Motif
                Sign = Positive
                XCommand =
```

FIGURE 7.3 – Exemple de résultats obtenus avec Ptjdej.

CKJM Extended

CKJM Extended [Jure 10] permet, à partir de bytecode Java, de calculer 19 métriques différentes pour chaque classe du logiciel. Les résultats sont affichés sur la sortie standard ou écrits dans un fichier. Il est possible d'extraire les données en XML. La figure 7.4 montre un exemple de données obtenues avec le logiciel CKJM Extended.

FIGURE 7.4 – Exemple de résultats obtenus avec CKJM Extended.

Commons Math

Commons Math [Comm 16] est une API Java développée par Apache. Elle est utilisée pour les calculs de corrélation et les tests statistiques.

7.5.2 Le logiciel créé pour cette étude : Eippoo

Afin de permettre la communication entre les différents logiciels tiers, ainsi qu'agréger et extraire les données utilisées pour l'étude, le logiciel Eippoo a été développé.

Eippoo permet :

- D'automatiser le processus de collecte des métriques sur plusieurs versions de plusieurs logiciels.
- D'automatiser le processus de détection des design patterns sur plusieurs versions de plusieurs logiciels.
- D'automatiser la détection des classes ayant subi l'introduction d'un pattern.
- D'assembler dans une même structure de données les résultats fournis par Ptjdej et ceux fournis par CKJM Extended.
- D'effectuer des analyses statistiques sur ces données.
- De générer des rapports utilisables avec Excel.

Eippoo n'a pas d'interface graphique ou d'interface en lignes de commandes. Il s'utilise en manipulant la classe principale dans Éclipse. Les données à fournir en entrée sont : une liste des design patterns à considérer et les répertoires contenant les sources des logiciels à analyser. La figure 7.5 montre un exemple d'arborescence à fournir en entrée à Eippoo.

En sortie, Eippoo génère un rapport qu'il est possible d'ouvrir avec un éditeur de texte ou directement avec Excel. La figure 7.6 montre un exemple de rapport qui a été généré.

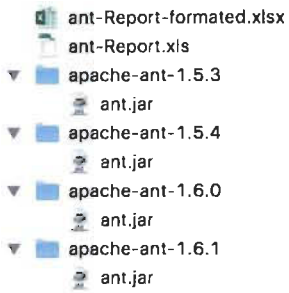

FIGURE 7.5 – Arborescence à analyser par Eippoo.

Numero de la version	0.5.6	0.6.0	0.7.0	0.7.1	0.7.2	0.7.3	0.7.4	0.8.0
Analyse des correlations								
rfc		0,37	0,32	0,32	0,33	0,33	0,34	0,30
wmc		0,32	0,29	0,28	0,30	0,30	0,31	0,27
cbo		0,31	0,25	0,24	0,24	0,24	0,23	0,21
ce		0,45	0,37	0,36	0,32	0,32	0,32	0,28
ca								
loc								
lcom				0,19	0,20	0,20	0,23	0,29
dit								
noc								
ttm		0,37	0,29	0,29	0,25	0,24	0,24	0,21
Impact suite à l'introduction d'un pattern								
rfc								
Nombre d'observations	124	124	124	112	112	111	104	97
Moyennes	46,7	49,8	50,6	54	54,9	55,9	60,9	62
TTest		-2,809945	-2,77548	-2,816398	-3,067304	-3,313319	-4,404647	-4,279988
wmc								
Nombre d'observations	124	124	124	112	112	111	104	97
Moyennes	17,3	19	19,5	20,9	21,4	21,8	24,1	24,8
TTest		-3,377189	-3,491821	-3,123849	-3,363653	-3,464304	-3,963224	-3,878813
cbo								
Nombre d'observations	124	124	124	112	112	111	104	97
Moyennes	12,8	13,5	13,6	14,7	14,8	15	16,3	16,4
TTest		-3,059741	-3,431985	-4,301253	-4,526928	-4,753533	-6,501833	-5,664328
ce								
Nombre d'observations	124	124	124	112	112	111	104	97
Moyennes	9,3	9,9	10	10,9	11	11	12,1	12
TTest		-2,451907	-2,741145	-3,672829	-3,780632	-3,922359	-5,783191	-5,385529
ca								
Nombre d'observations	124	124	124	112	112	111	104	97
Moyennes	3,7	3,8	3,8	4	4,1	4,2	4,4	4,7
TTest				-2,446469	-2,933053	-3,202793	-3,296204	-2,48063

FIGURE 7.6 – Rapport Excel généré par Eippoo.

7.6 Présentation des études de cas

De nombreux logiciels ont été soumis (au début) à l'étude. Pour la plupart d'entre eux, il n'y avait pas suffisamment de design patterns détectés. Au final, cinq d'entre eux ont été sélectionnés. La table 7.1 regroupe les informations sur ces logiciels. Voici quelques précisions sur chaque colonne de cette table :

Développeurs : L'organisation qui développe le logiciel.

Nb versions : Le nombre de versions différentes du logiciel.

Age : La durée de vie du logiciel.

Nb classes : Le nombre de classes de l'ensemble des versions du logiciel.

URL des JARs : L'URL du dépôt qui met à disposition les JARs des différentes versions. La validité des URL a été vérifiée pour la dernière fois en novembre 2015.

JfreeChart	
JFreeChart est une API Java, gratuite, permettant la création de graphiques à partir de données. Par exemple, des histogrammes ou des nuages de points.	
Développeurs	Object Refinery
Nb versions	50
Age	13 ans
Nb classes	18 981
URL des JARs	sourceforge.net/projects/jfreechart/files/

TABLE 7.1 – Sujets de l'étude - JFreeChart.

Ant	
Ant permet l'automatisation des opérations répétitives du développement logiciel telles que la compilation, la génération de Javadoc ou la compression au format JAR.	
Développeurs	Apache
Nb versions	22
Age	12 ans
Nb classes	18 628
URL des JARs	archive.apache.org/dist/ant/source/

TABLE 7.2 – Sujets de l'étude - Ant.

MySQL	
MySqlConnector est une API Java qui fournit des drivers de connexion aux bases de données.	
Développeurs	Oracle
Nb versions	46
Age	13 ans
Nb classes	11 184
URL des JARs	mvnrepository.com/artifact/mysql/mysql-connector-java

TABLE 7.3 – Sujets de l'étude - MySQL.

Eclipse	
Eclipse JDT est le noyau central du célèbre environnement de développement Eclipse.	
Développeurs	Eclipse Foundation
Nb versions	43
Age	14 ans
Nb classes	57 846
URL des JARs	archive.eclipse.org/eclipse/downloads

TABLE 7.4 – Sujets de l'étude - Eclipse.

Chapitre 8

Etude de cas

Le chapitre 8 présente les résultats obtenus suite à l'application du processus de collecte décrit dans la section 7.4, avec les outils répertoriés dans la section 7.5, sur les logiciels présentés dans la section 7.6. L'objectif est d'apporter des réponses aux Questions 1' et 2' posées dans la section 7.2.

Ce chapitre est divisé en deux sections. La section 8.1 regroupe et étudie les résultats relatifs aux Questions 1'. La section 8.2 regroupe et étudie les résultats relatifs aux Questions 2'. Ces deux sections ont la même structure :

1. Rappel des questions de recherche.
2. Hypothèses.
3. Résultats.
4. Évaluation des hypothèses.
5. Réponses aux questions de recherche.

8.1 Étude des Questions 1.1' et 1.2'

8.1.1 Rappel des questions de recherche

Rappel des questions de recherche 1.1' et 1.2'.

Question 1.1' : Est-ce que l'introduction d'un design pattern sur une classe provoque un changement significatif des valeurs des métriques de testabilité ?

Question 1.2' : Est-ce que l'introduction d'un design pattern sur une classe provoque un changement significatif des valeurs des métriques de changeabilité ?

8.1.2 Hypothèses

À partir des questions de recherches 1.1' et 1.2', nous posons les hypothèses de recherche.

Considérons v et $v+1$ les valeurs des métriques avant et après l'introduction.

Hypothèse	Résumé	Type
H0 $v = v + 1$	Il n'y a pas de changement de la valeur des métriques après l'introduction.	Null
H1 $v \neq v + 1$	Il y a un changement de la valeur des métriques sans tendance déterminable.	Alternative
H2 $v < v + 1$	Il y a une augmentation de la valeur des métriques après l'introduction.	Alternative
H3 $v > v + 1$	Il y a une diminution de la valeur des métriques après l'introduction.	Alternative

TABLE 8.1 – Hypothèses Questions 1.1' et 1.2'.

8.1.3 Résultats

Les tableaux suivants présentent les résultats des "paired TTests" entre les valeurs des métriques avant l'introduction d'un design pattern et les valeurs des métriques après l'introduction d'un design pattern.

MySqlConnector n'apparaît pas dans les résultats car il n'a pas une quantité significative de cas d'introduction pour effectuer les TTests.

La colonne TTests indique la force de la différence entre les métriques avant et après l'introduction. Une valeur positive signifie une augmentation de la valeur des métriques et une valeur négative signifie une diminution de la valeur des métriques. La colonne PValues est la PValue associée aux TTests. Une PValue supérieur à 0,05 signifie que le test n'est pas statistiquement significatif. Les résultats des tests ayant des PValues supérieures à 0,05 sont en gris claire.

Métriques	Moyennes avant	Moyennes après	TTests	PValues
RFC	46,7	49,8	2,81	0,01
WMC	17,3	19,0	3,38	$< 0{,}01$
CBO	12,8	13,5	3,06	$< 0{,}01$
CE	9,3	9,9	2,45	0,02
NA	3,7	3,8	1,35	0,18
LOC	506,1	539,9	2,93	$< 0{,}01$
LCOM	193	256,9	2,47	0,01
DIT	2,7	2,7	1,00	0,32
NOC	0,6	0,7	2,14	0,03
TTM	506,1	521,2	1,87	0,06
Nombre d'observations : 124 introductions				

TABLE 8.2 – JFreeChart - Impact de l'introduction d'un design pattern.

Métriques	Moyennes avant	Moyennes après	TTests	PValues
RFC	34,4	35,4	1,41	0,16
WMC	11,1	11,1	0,05	0,96
CBO	9,0	9,8	1,96	0,05
CE	6,2	6,7	2,77	0,01
CA	3,1	3,5	1,15	0,25
LOC	285,2	283,8	-0,19	0,85
LCOM	87,3	109,2	3,95	$< 0,01$
DIT	2,6	3,5	9,77	$< 0,01$
NOC	0,7	0,8	2,30	0,02
TTM	403,4	507,7	12,06	$< 0,01$
Nombre d'observations : 232 introductions				

TABLE 8.3 – Ant - Impact de l'introduction d'un design pattern.

Métriques	Moyennes avant	Moyennes après	TTests	PValues
RFC	38,1	56,1	10,34	$< 0,01$
WMC	13,4	20,6	9,20	$< 0,01$
CBO	27,0	38,8	8,40	$< 0,01$
CE	13,2	19,8	11,18	$< 0,01$
CA	15,4	22,6	6,71	$< 0,01$
LOC	447,3	979,8	8,56	$< 0,01$
LCOM	245,6	433,7	2,80	$< 0,01$
DIT	3,0	2,9	-2,15	0,02
NOC	0,9	1,3	4,01	$< 0,01$
TTM	721	947,5	9,75	$< 0,01$
Nombre d'observations : 475 introductions				

TABLE 8.4 – JDT - Impact de l'introduction d'un design pattern.

8.1.4 Évaluation des hypothèses

Sous l'angle de la changeabilité puis sous l'angle de la testabilité, en s'appuyant sur les résultats de la section précédente, les hypothèses proposées dans la section 8.1.2 sont évaluées.

Dans un premier temps, est évaluée l'hypothèse nulle H0. Cette hypothèse peut être rejetée si, pour la plupart des métriques de changeabilité ou de testabilité, les PValues associées au TTest sont inférieures ou égales à 0.05.

Dans un second temps, si l'hypothèse nulle est rejetée, les hypothèses alternatives seront testées. Les hypothèses alternatives considèrent la tendance de la différence entre v et $v+1$ (i.e., une augmentation ou une diminution de la valeur des métriques). La valeur des résultats des TTest permet de connaitre la tendance de la différence. Si la valeur est positive, il y a une augmentation de la valeur des métriques entre v et $v+1$. Si la valeur est négative, il y a une diminution de la valeur des métriques entre v et $v+1$.

Hypothèses relatives à la testabilité

Test de l'hypothèse nulle

JfreeChart	PValue ≤ 0.05 pour 7 métriques de testabilité sur 9
Ant	PValue ≤ 0.05 pour 6 métriques de testabilité sur 9
JDT	PValue ≤ 0.05 pour 9 métriques de testabilité sur 9

Ces résultats permettent de rejeter l'hypothèse nulle $v = v+1$

Test des hypothèses alternatives

Pour les trois logiciels, exception faite de la DIT pour JDT, la valeur de tous les résultats des TTest relatifs aux métriques de testabilité est positive, donc à chaque fois $v < v+1$.

Ces résultats permettent de valider l'hypothèse H2. Il y a donc une augmentation de la valeur des métriques de testabilité après l'introduction d'un design pattern.

Hypothèses relatives à la changeabilité

Test de l'hypothèse nulle

JfreeChart	PValue ≤ 0.05 pour 5 métriques de changeabilité sur 6
Ant	PValue ≤ 0.05 pour 3 métriques de changeabilité sur 6
JDT	PValue ≤ 0.05 pour 6 métriques de changeabilité sur 6

Ces résultats permettent de rejeter l'hypothèse nulle $v = v + 1$

Test des hypothèses alternatives

Pour les trois logiciels, la valeur de tous les résultats des TTest relatifs aux métriques de changeabilité est positive, donc à chaque fois $v < v + 1$.

Ces résultats permettent de valider l'hypothèse H2. Il y a donc une augmentation de la valeur des métriques de changeabilité après l'introduction d'un design pattern.

8.1.5 Réponses aux Questions 1.1' et 1.2'

En se basant sur les hypothèses évaluées dans la section précédente, il est possible de répondre aux Questions 1.1' et 1.2'

Question 1.1' : Est-ce que l'introduction d'un design pattern sur une classe provoque un changement significatif des valeurs des métriques de testabilité ?

Réponse 1.1' : La validité de l'hypothèse H2 relative à la testabilité (section 8.1.4) permet de répondre que l'introduction d'un design pattern sur une classe, provoque un changement significatif sur les métriques de testabilité. De plus ce changement consiste en une augmentation de la valeur de ces métriques. L'introduction d'un design pattern sur une classe provoque donc une augmentation significative de la valeurs des métriques de testabilité de cette classe. En fonction des métriques considérées, une augmentation de leur valeur signifie que la classe est plus complexe, plus couplée, d'une taille plus importante, etc. C'est pourquoi, cette augmentation de la valeur des métriques est le signe d'une dégradation de la testabilité de la classe.

Question 1.2' : Est-ce que l'introduction d'un design pattern sur une classe provoque un changement significatif des valeurs des métriques de changeabilité ?

Réponse 1.2' : La validité de l'hypothèse H2 relative à la changeabilité (section 8.1.4) permet de répondre que l'introduction d'un design pattern sur une classe, provoque un changement significatif sur les métriques de changeabilité. De plus ce changement consiste en une augmentation de la valeur de ces métriques. L'introduction d'un design pattern sur une classe provoque donc une augmentation significative de la valeurs des métriques de changeabilité de cette classe. En fonction des métriques considérées, une augmentation de leur valeur signifie que la classe est plus complexe, plus couplée, d'une taille plus importante, etc. C'est pourquoi, cette augmentation de la valeur des métriques est le signe d'une dégradation de la changeabilité de la classe.

8.2 Étude des Questions 2.1' et 2.2'

8.2.1 Rappel des questions de recherche

Question 2.1' : Pour l'ensemble des classes d'une version, est-ce que la valeur des métriques de testabilité des classes est corrélée au nombre de design patterns dans lequel elles sont impliquées ?

Question 2.2' : Pour l'ensemble des classes d'une version, est-ce que la valeur des métriques de changeabilité des classes est corrélée au nombre de design patterns dans lequel elles sont impliquées ?

8.2.2 Hypothèses

Les variables suivantes sont utilisées pour définir les hypothèses ci-dessous relatives aux questions de recherche 2.1' et 2.2' :

M : la valeurs des Métriques des classes.

N : le Nombre d'implications des classes dans des design patterns.

$P(M, N)$: La PValue du test de corrélation entre M et N.

Hypothèses	**Résumés**	**Types**
H0 $P(M, N) > .05$	La corrélation observée entre M et N est due au hasard	Null
H1 $P(M, N) \leq .05$	Il existe une corrélation entre M et N	Alternative

TABLE 8.5 – Hypothèses Questions 2.1' et 2.2'.

8.2.3 Résultats

Ci-dessous, les figures 8.1, 8.2, 8.3 et 8.4 présentent les résultats d'une analyse de corrélation entre les valeurs des métriques des classes et le nombre de design patterns dans lequel ces classes sont impliquées.

La colonne "Versions" répertorie les versions analysées. Chaque ligne du tableau correspond à l'analyse d'une version. Les résultats dans les colonnes centrales "Métriques" sont les résultats des analyses de corrélation pour chaque métrique. Quand la PValue associée à ces valeurs est supérieure à 0,05 (i.e., les résultats ne sont pas significatifs), les résultats sont barrés et affichés en gris clair. Les dernières colonnes représentent le nombre de classes par implication. Les colonnes "0 dp", "1 dp", "2 dp" ou "3 dp" correspondent au nombre de classes qui sont impliquées dans 0, 1, 2 ou 3 design pattern(s).

Versions	Métriques										Nb de classe	
	rfc	wmc	cbo	ce	ca	loc	lcom	dit	noc	ttm	0 dp	1 dp
1.5.3					0,18						397	10
1.5.4					0,18						397	10
1.6.0					0,14						507	16
1.6.1					0,14						508	16
1.6.2					0,14						529	24
1.6.3					0,14						551	25
1.6.4					0,14						551	25
1.6.5					0,14						551	25
1.7.0	0,38	0,33	0,15	0,44	0,03	0,22	0,17	0,54	0,10	0,59	503	249
1.7.1	0,37	0,32	0,15	0,44	0,02	0,22	0,16	0,56	0,09	0,59	513	256
1.8.0	0,37	0,32	0,15	0,44	0,03	0,23	0,17	0,57	0,10	0,61	596	274
1.8.1	0,37	0,31	0,15	0,44	0,03	0,23	0,17	0,57	0,10	0,60	597	276
1.8.2	0,36	0,32	0,12	0,44	0,01	0,22	0,16	0,62	0,08	0,63	724	366
1.8.3	0,36	0,32	0,12	0,44	0,01	0,23	0,16	0,61	0,08	0,62	727	366
1.8.4	0,36	0,32	0,12	0,44	0,01	0,23	0,16	0,61	0,08	0,62	728	366
1.9.0	0,35	0,31	0,12	0,44	0,01	0,22	0,16	0,61	0,08	0,62	748	368
1.9.1	0,35	0,31	0,12	0,43	0,01	0,22	0,16	0,61	0,08	0,62	757	373
1.9.2	0,35	0,31	0,12	0,43	0,01	0,22	0,15	0,61	0,08	0,62	758	373
1.9.3	0,35	0,31	0,12	0,43	0,01	0,22	0,15	0,61	0,08	0,62	762	373
1.9.4	0,35	0,31	0,12	0,44	0,01	0,22	0,16	0,61	0,09	0,62	767	374
1.9.5	0,35	0,30	0,12	0,44	0,01	0,22	0,15	0,61	0,09	0,62	770	376
1.9.6	0,35	0,30	0,12	0,44	0,01	0,22	0,15	0,61	0,09	0,62	770	376

FIGURE 8.1 – Ant - Corrélations entre les métriques et le nombre d'implications dans un design pattern des classes.

Versions	Métriques										Nb de classes		
	rfc	wmc	cbo	ce	ca	loc	lcom	dit	noc	ttm	0 dp	1 dp	2 dp
0.5.6			0,12		0,11				-0,11		82	8	0
0.6.0			0,31		-0,10				-0,12		72	11	0
0.7.0					-0,07				-0,09		89	11	0
0.7.1					-0,04				-0,08		102	15	0
0.7.2					-0,01				-0,09		102	13	0
0.7.3									-0,08		103	13	0
0.7.4									-0,08		105	13	0
0.8.0											108	12	0
0.8.1											146	7	0
0.9.0									-0,03		118	7	0
0.9.1											118	7	0
0.9.2											122	7	0
0.9.3											210	30	0
0.9.4											226	31	0
0.9.5											277	32	0
0.9.6											279	32	0
0.9.7											297	33	4
0.9.8					-0,06						299	34	4
0.9.9					-0,04						303	40	5
0.9.10					0,06						324	11	4
0.9.11					0,06				0,01		337	11	4
0.9.12					0,10				0,09		311	45	10
0.9.13					0,10				0,09		317	45	10
0.9.14					0,10				0,05		338	35	10
0.9.15					0,09				0,06		341	41	10
0.9.16					0,09				0,06		349	41	10
0.9.17					0,06				0,08		338	66	9
0.9.18					0,06				0,08		343	67	9
0.9.19					0,05				0,09		354	69	9
0.9.20					0,05				0,09		355	69	9
0.9.21					0,06				0,07		368	67	9
1.0.0					0,01				0,06		371	97	13
1.0.1					0,01				0,06		372	97	13
1.0.2					0,01				0,05		370	102	13
1.0.3					0,00				0,04		388	108	13
1.0.4					0,00				0,04		395	111	13
1.0.5	0,51	0,44	0,28	0,46	0,00	0,46	0,33	0,39	0,01	0,46	399	109	16
1.0.6	0,52	0,44	0,29	0,47	0,00	0,47	0,32	0,39	0,00	0,47	408	109	18
1.0.7	0,52	0,44	0,29	0,47	0,01	0,47	0,32	0,38	0,00	0,46	429	114	19
1.0.8	0,52	0,44	0,29	0,47	0,01	0,47	0,32	0,38	0,00	0,47	429	114	19
1.0.9	0,52	0,44	0,29	0,47	0,01	0,47	0,32	0,38	0,00	0,47	429	114	19
1.0.10	0,52	0,44	0,28	0,46	0,01	0,47	0,32	0,38	0,00	0,46	432	117	19
1.0.11	0,51	0,43	0,29	0,45	0,02	0,45	0,31	0,39	0,01	0,46	441	124	19
1.0.12	0,51	0,43	0,29	0,45	0,02	0,45	0,31	0,39	0,01	0,46	443	124	19
1.0.13	0,49	0,42	0,28	0,44	0,02	0,43	0,31	0,39	-0,01	0,45	465	127	19
1.0.14	0,49	0,42	0,28	0,44	0,02	0,44	0,32	0,38	-0,01	0,45	473	127	20
1.0.15	0,49	0,42	0,24	0,44	0,01	0,44	0,32	0,35	-0,01	0,44	477	127	20
1.0.16	0,47	0,41	0,24	0,44	0,01	0,42	0,30	0,36	-0,01	0,44	480	128	19
1.0.17	0,47	0,41	0,24	0,44	0,01	0,42	0,30	0,35	-0,01	0,44	483	128	19
1.0.19	0,47	0,41	0,24	0,44	0,01	0,42	0,31	0,36	-0,01	0,45	490	129	19

FIGURE 8.2 – JFreeChart - Corrélations entre les métriques et le nombre d'implications dans un design pattern des classes.

Versions	Métriques										Nb de classes	
	rfc	wmc	cbo	ce	ca	loc	lcom	dit	noc	ttm	0 dp	1 dp
3.1.11								-0,06			125	5
3.1.12								-0,06			127	5
3.1.13								-0,05			128	5
3.1.14								-0,05			128	5
5.0.2								-0,02			142	10
5.0.3								-0,02			142	10
5.0.4								0,00			143	9
5.0.5								0,00			143	9
5.0.7								0,00			149	9
5.0.8								0,00			150	9
5.1.1								-0,05			190	15
5.1.2								-0,05			190	15
5.1.3								-0,06			192	15
5.1.4								-0,03			206	12
5.1.5								-0,03			206	12
5.1.6								-0,03			213	12
5.1.8								-0,02			215	12
5.1.9								-0,02			215	12
5.1.10								-0,02			215	12
5.1.11								-0,02			219	12
5.1.12								-0,02			219	12
5.1.13							0,38	-0,07			225	17
5.1.14							0,37	-0,08			225	18
5.1.15							0,37	-0,08			226	18
5.1.16							0,38	0,07			228	17
5.1.17							0,38	-0,07			228	17
5.1.18							0,38	-0,07			228	17
5.1.19							0,38	-0,07			232	17
5.1.20							0,38	-0,07			232	17
5.1.21								-0,06			244	17
5.1.22								-0,06			245	17
5.1.23								-0,06			248	17
5.1.24	0,51	0,40	0,50	0,64	0,27	0,47	0,31	-0,07	0,08	0,10	248	18
5.1.25	0,51	0,40	0,50	0,64	0,27	0,47	0,31	-0,07	0,08	0,09	248	18
5.1.26	0,51	0,40	0,49	0,64	0,27	0,47	0,31	-0,07	0,08	0,09	251	18
5.1.27	0,32	0,22	0,50	0,63	0,29	0,46	0,05	-0,06	0,07	0,26	261	18
5.1.28	0,32	0,22	0,50	0,63	0,29	0,46	0,05	-0,06	0,07	0,26	261	18
5.1.29	0,32	0,22	0,50	0,63	0,29	0,45	0,05	-0,06	0,07	0,26	261	18
5.1.30	0,28	0,20	0,48	0,59	0,29	0,43	0,05	-0,05	0,07	0,25	306	19
5.1.31	0,28	0,21	0,48	0,59	0,29	0,43	0,05	-0,05	0,07	0,25	309	19
5.1.32	0,28	0,21	0,47	0,59	0,28	0,43	0,05	-0,05	0,07	0,25	310	19
5.1.33	0,28	0,21	0,47	0,59	0,27	0,44	0,05	-0,05	0,07	0,25	310	19
5.1.34	0,28	0,20	0,47	0,59	0,27	0,44	0,05	-0,05	0,07	0,25	311	19
5.1.35	0,44	0,36	0,45	0,58	0,25	0,45	0,29	-0,06	0,09	0,08	313	20
5.1.36	0,44	0,36	0,45	0,58	0,25	0,45	0,29	-0,06	0,09	0,08	313	20
5.1.37	0,44	0,36	0,45	0,57	0,25	0,45	0,29	-0,06	0,10	0,07	321	20

FIGURE 8.3 – MysqlConnector - Corrélations entre les métriques et le nombre d'implications dans un design pattern des classes.

Versions	Métriques										Nb de classes			
	rfc	wmc	cbo	ce	ca	loc	lcom	dit	noc	ttm	0 dp	1 dp	2 dp	3 dp
1.0											645	162	0	0
2.0											654	242	8	0
2.0.1											654	242	8	0
2.0.2											654	241	9	0
2.1											713	250	13	0
2.1.1											713	249	13	0
2.1.2											715	249	13	0
2.1.3											719	249	13	0
3.0											837	200	57	30
3.0.1											837	200	57	30
3.0.2											837	200	57	30
3.1											877	292	43	43
3.1.1											878	292	43	43
3.1.2											878	292	43	43
3.2											959	301	41	44
3.2.2											960	301	41	44
3.3											971	323	42	44
3.3.1											970	323	42	44
3.3.2											971	323	42	44
3.4											1026	338	42	44
3.4.1											1027	338	42	44
3.4.2											1029	338	42	44
3.5											1062	327	42	44
3.5.1											1062	327	42	44
3.5.2											1064	327	42	44
3.6.1											1061	324	42	44
3.6.2											1061	324	42	44
3.7											1064	326	42	44
3.7.1											1067	330	43	44
3.7.2	0,29	0,22	0,28	0,29	0,19	0,15	0,07	0,22	0,17	0,32	1068	330	43	44
3.8	0,29	0,22	0,28	0,30	0,19	0,15	0,07	0,23	0,17	0,33	1080	336	43	44
3.8.2	0,29	0,22	0,28	0,30	0,19	0,15	0,07	0,23	0,17	0,33	1080	336	43	44
4.2	0,29	0,22	0,28	0,30	0,19	0,15	0,07	0,23	0,17	0,33	1080	336	43	44
4.2.1	0,29	0,22	0,28	0,30	0,19	0,15	0,07	0,23	0,17	0,33	1080	336	43	44
4.2.2	0,29	0,22	0,28	0,30	0,19	0,15	0,07	0,23	0,17	0,33	1080	336	43	44
4.3	0,30	0,21	0,28	0,31	0,17	0,15	0,06	0,24	0,16	0,35	1078	334	53	44
4.3.1	0,30	0,21	0,28	0,31	0,17	0,15	0,06	0,24	0,16	0,35	1078	334	53	44
4.3.2	0,30	0,21	0,28	0,31	0,17	0,15	0,06	0,24	0,16	0,35	1078	334	53	44
4.4	0,33	0,22	0,35	0,36	0,23	0,18	0,05	0,31	0,18	0,41	1129	294	148	49
4.4.1	0,33	0,22	0,35	0,36	0,23	0,18	0,05	0,31	0,18	0,41	1129	294	148	49
4.4.2	0,33	0,22	0,35	0,36	0,23	0,18	0,05	0,31	0,18	0,41	1132	294	148	49
4.5	0,34	0,23	0,36	0,37	0,24	0,20	0,05	0,31	0,18	0,42	1193	296	150	49
4.5.1	0,34	0,23	0,36	0,37	0,24	0,20	0,05	0,31	0,18	0,42	1193	296	150	49

FIGURE 8.4 – JDT - Corrélations entre les métriques et le nombre d'implications dans un design pattern des classes.

8.2.4 Évaluation des hypothèses

L'hypothèse nulle H0, peut être rejetée, si, pour la plupart des métriques de changeabilité ou de testabilité, les PValues associées aux résultats des tests de corrélation sont inférieures ou égales à 0.05.

Considérant les résultats de la section précédente, il n'est pas possible de rejeter ou de valider l'hypothèse nulle pour l'ensemble des versions. En effet, il est possible de constater qu'il existe un schéma dans les résultats qui est commun aux quatre logiciels.

Pour environ la première moitié des versions, c'est à dire les versions les plus anciennes, $P(M, N) > .05$, les PValues des résultats des corrélations entre M et N démontrent que l'hypothèse nulle H0 est acceptée. Pour ces versions, il n'existe donc pas de corrélations entre la valeur des métriques des classes et leur nombre d'implications dans des design patterns.

En revanche pour la seconde moitié des versions, les versions les plus récentes, $P(M, N) \leq .05$, les PValues des résultats des corrélations entre M et N démontrent que l'hypothèse null H0 peut être rejetée et que l'hypothèse alternative H1 est donc acceptée. Pour ces versions, il existe donc une corrélation entre la valeur des métriques de changeabilité et de testabilité des classes et leur nombre d'implications dans des design patterns.

8.2.5 Réponses aux Questions 2.1' et 2.1'

En se basant sur les hypothèses testées dans la section précédente, il est possible d'apporter des éléments de réponse aux Questions 2'

Question 2.1' : Pour l'ensemble des classes d'une version, est-ce que la valeur des métriques de testabilité des classes est corrélée au nombre de design patterns dans lequel elles sont impliquées ?

Réponse 2.1' : Considérant que l'hypothèse nulle H0 ne peut être rejetée que pour les versions les plus récentes, la réponse à cette question est mitigée. Les résultats suggèrent qu'après un certain nombre de versions, la valeur des métriques de testabilité d'une classe deviendrait corrélée au nombre d'implications dans des design patterns de cette classe. Donc, après un certain nombre de versions, la valeur des métriques de testabilité des classes qui sont impliquées dans un ou plusieurs design patterns est supérieure à la valeur des métriques de testabilité des classes qui ne sont pas impliquées dans un design pattern (plus les valeurs des métriques sont élevées moins bonne est la testabilité).

Question 2.2' : Pour l'ensemble des classes d'une version, est-ce que la valeur des métriques de changeabilité des classes est corrélée au nombre de design patterns dans lequel elles sont impliquées ?

Réponse 2.2' : Considérant que l'hypothèse nulle H0 ne peut être rejetée que pour les versions les plus récentes, la réponse à cette question est aussi discutable. Les résultats suggèrent qu'après un certain nombre de versions, la valeur des métriques de changeabilité d'une classe deviendrait corrélée au nombre d'implications dans des design patterns de cette classe. Donc, après un certain nombre de versions, la valeur des métriques de changeabilité des classes qui sont impliquées dans un ou plusieurs design patterns est supérieure à la valeur des métriques de changeabilité des classes qui ne sont pas impliquées dans un design pattern (plus les valeurs des métriques sont élevées moins bonne est la changeabilité).

8.3 Conclusion

Au travers de ce Chapitre 8, une partie des données collectées lors de ce travail de recherche a été présentée.

Ces données permettent d'affirmer que pour les logiciels étudiés, l'utilisation des design patterns implique une dégradation de la testabilité et la changeabilité des classes.

Cette conclusion est induite par deux constats qui découlent des résultats précédents, à savoir :

- Suite à l'introduction d'un design pattern, les classes affectées perdent en testabilité et en changeabilité.
- Après un certain nombre de versions, une corrélation positive apparait entre le nombre d'implications d'une classe dans des design patterns et ses métriques de testabilité / changeabilité. Autrement dit, après un certain nombre de versions, une classe impliquée dans un ou plusieurs design patterns a statistiquement une moins bonne testabilité / changeabilité que les classes qui ne sont pas impliquées dans un design pattern.

Démontrer que l'implication d'une classe dans un ou plusieurs design patterns implique une dégradation de la testabilité et la changeabilité de cette classe, ne signifie pas que les design patterns sont la cause de cette dégradation.

En effet, lors d'une introduction d'un design pattern, deux scénarios sont possibles : Soit cette introduction est effectuée dans le cadre d'un refactoring, soit cette introduction est effectuée dans le cadre d'un ajout de fonctionnalités au programme. Il est probable, bien que ce soit difficilement vérifiable,

que la majorité des cas d'introduction d'un design pattern soit effectuée dans le cadre d'un ajout de fonctionnalité.

Dans le contexte d'un ajout de fonctionnalité, étant donné que la classe existait déjà sans être impliquée dans un design pattern, l'implication de cette classe dans un design pattern nécessite une adaptation de cette dernière. Cette adaptation pour intégrer le design pattern entraine une perte de testabilité et de changeabilité.

Bien que l'adaptation de la classe peut entrainer une perte de testabilité et de changeabilité, il peut être supposé que l'implication de cette classe dans un design pattern lui permette par la suite de mieux s'adapter au changement. Dans ce cas, cette perte immédiate de changeabilité et de testabilité évoquée dans le premier constat, permettrait d'éviter des baisses plus importantes par la suite.

Cependant, le second constat vient remettre en question cette supposition. En effet, ce constat stipule qu'après un certain nombre de versions, une classe impliquée dans un ou plusieurs design pattern a statistiquement une moins bonne testabilité / changeabilité que les classes qui ne sont pas impliquées dans un design pattern.

Deux explications sont possibles pour expliquer cette différence :

La première explication, à charge contre les design patterns, pourrait leur imputer la responsabilité de la dégradation de la testabilité et de la changeabilité. Au fur et à mesure des versions, les changements dégraderaient davantage la testabilité et la changeabilité quand ils concernent des classes qui sont impliquées dans un ou plusieurs design patterns.

La seconde explication, à la décharge pour les design patterns, pourrait être que ceux-ci sont utilisés dans les parties les plus complexes des applications. Cette complexité entrainerait de facto une difficulté supplémentaire pour tester et modifier les classes. Donc, les classes qui ne sont pas impliquées

dans un design pattern auraient une meilleure testabilité / changeabilité que les classes qui sont impliquées dans un design pattern car ces dernières se situent dans les parties les plus complexes de l'application.

Quatrième partie

Discussions et conclusion

Chapitre 9

Discussions et conclusion

9.1 Biais possibles de cette étude

Détection des design patterns

Lors de cette étude, la principale difficulté a été de réussir à détecter des occurences de design patterns dans du code source. Cette détection a été réalisée grâce à l'outil Ptidej [Gueh 08]. Afin d'obtenir des résultats les plus fiables possible, seuls les 4 design patterns qui sont détectés par l'outil avec la meilleure fiabilité ont été utilisés. Bien que représentant les trois catégories des patterns GOF, il est possible que ces 4 design patterns ne soient pas représentatifs de l'ensemble des design patterns. De plus, ce doute sur la représentativité des 4 patterns augmente si l'on considère l'ensemble des design patterns et pas uniquement ceux du GOF. À notre connaissance, il n'existe pas aujourd'hui d'outils similaires à Ptidej qui permettraient de diversifier les patterns détectés.

Évaluation de la testabilité et de la changeabilité

Les cadres d'évaluation de la testabilité et de la changeabilité sont variés et parfois contradictoires. Une autre approche d'évaluation de la testabilité et de la changeabilité pourrait faire varier les résultats.

Variété des sujets de l'étude

L'ensemble des versions de quatre logiciels open source écrits en JAVA a été utilisé comme sujet d'étude. Néanmoins, ces logiciels ne sont pas représentatifs de l'ensemble des logiciels. Par exemple, des résultats différents pourraient être observés sur des logiciels écrits dans d'autres languages que JAVA ou sur des logiciels qui ne sont pas open source.

9.2 Futurs travaux

Variété des design patterns

Un bon point d'amélioration serait d'étendre la variété des design patterns considérés, en incluant par exemple d'autres patterns GOF. Pour étendre la variété des design patterns considérés, il est nécessaire de réussir à obtenir des cas concrets d'applications de ces patterns dans des projets. La détection de ces cas concrets nécessiterait l'aide d'un outil similaire à Ptidej. Une alternative à la création d'un outil d'analyse automatique pourrait être une approche plus collaborative, par exemple en demandant la participation de plusieurs groupes de développement (une communauté open source) qui incluraient des balises dans leur code pour identifier les design patterns qu'ils utilisent.

Variété des sujets d'études

Un autre point d'amélioration pourait être de considérer des logiciels qui ne sont pas open source ou des logiciels écrits dans d'autres langages de programmation, tel le C#.

Approfondir l'analyse 1

D'après les résultats présentés précédemment, les design patterns sont liés à une baisse de la testabilité et de la changeabilité. Ces conclusions sont induites par des résultats statistiques, ce qui veut dire, que pour la majorité des classes, les design patterns sont liés à une baisse de la testabilité et de la changeabilité. Néanmoins, pour certaines classes les design patterns sont liés à une amélioration de la testabilité et de la changeabilité.

Il pourrait donc être intéressant d'étudier les différences qui peuvent exister entre les cas où les design patterns sont liés à une amélioration de la testabiltié/changeabilité et les cas où les design patterns sont liés à une détérioration de la testabilité/changeabilité. Ces différences pourraient donner des indices sur les facteurs qui conditionneraient une bonne utilisation des design patterns.

Approfondir l'analyse 2

Une théorie apportée dans l'analyse des résultats prétend que les design patterns sont utilisés dans les parties les plus complexes des applications (ce qui expliquerait la moins bonne testabilité/changeabilité des classes qui les composent). Il serait intéressant d'approfondir cette théorie afin de confirmer que les design patterns sont effectivement utilisés dans les zones les plus complexes.

9.3 Conclusion

Cette étude a porté sur l'évaluation de l'impact des design patterns sur la testabilité et la changeabilité.

L'ensemble des versions de quatre logiciels écrits en Java ont été analysées (pour un total de 106 639 classes réparties dans 161 versions). Ces analyses ont permis de collecter, au travers de différentes métriques, des données sur la testabilité et la changeabilité des classes. Ces analyses ont également permis d'obtenir des données relatives à l'implication des classes dans certains design patterns.

À partir des données collectées, des analyses statistiques ont été réalisées pour déterminer la relation qui existe entre les design patterns et la testabilité/changeabilité. Les résultats démontrent que l'implication des classes dans un ou plusieurs design patterns est liée à une dégradation de la testabilité et de la changeabilité. Cependant, ces résultats ne permettent pas de conclure sur le sens de cette relation. Est-ce l'implication d'une classe dans un design pattern qui entraine une baisse de la testabilité / changeabilité ou est-ce l'inverse ?

Bibliographie

[Abdu 14] D. Abdullah, R. Srivastava, and M. Khan. "Modifiability : A Key Factor To Testability". *International Journal of Advanced Information Science and Technology*, Vol. 26, No. 26, pp. 62–71, 2014.

[Abdu 15a] M. H. K. Abdullah and R. Srivastava. "Testability Measurement Model for Object Oriented Design (TMMOOD)". *ArXiv e-prints*, March 2015.

[Abdu 15b] M. H. K. Abdullah, Dr Reena Srivastava. "Flexibility : A Key Factor To Testability". *International journal of Software Engineering & Applications (IJSEA)*, Vol. 6, No. 1, pp. 89–99, 2015.

[Ager 98] E. Agerbo and A. Cornils. "How to Preserve the Benefits of Design Patterns". *SIGPLAN Not.*, Vol. 33, No. 10, pp. 134–143, Oct. 1998.

[Ajrn 00] M. Ajrnal Chaumun, H. Kabaili, R. Keller, F. Lustman, and G. Saint-Denis. "Design properties and object-oriented software changeability". In : *Software Maintenance and Reengineering, 2000. Proceedings of the Fourth European*, pp. 45–54, Feb 2000.

[Ajrn 99] M. Ajrnal Chaumun, H. Kabaili, R. Keller, and F. Lustman. "A change impact model for changeability assessment in object-oriented software systems". In : *Software Maintenance and Reengineering, 1999. Proceedings of the Third European Conference on*, pp. 130–138, 1999.

[Alex 77] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language : Towns, Buildings, Construction.* Oxford University Press, New York, August 1977.

[Alsh 11] M. Alshayeb. "The impact of refactoring to patterns on software quality attributes". *Arabian Journal for Science and Engineering*, Vol. 36, No. 7, pp. 1241–1251, 2011.

[Amou 06] M. Amoui, S. Mirarab, S. Ansari, and C. Lucas. "A genetic algorithm approach to design evolution using design pattern transformation". *International Journal of Information Technology and Intelligent Computing*, Vol. 1, No. 2, pp. 235–244, 2006.

[Ampa 12] A. Ampatzoglou, G. Frantzeskou, and I. Stamelos. "A methodology to assess the impact of design patterns on software quality". *Information and Software Technology*, Vol. 54, No. 4, pp. 331 – 346, 2012.

[Anto 98a] G. Antoniol, R. Fiutem, and L. Cristoforetti. "Design pattern recovery in object-oriented software". In : *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*, pp. 153–160, 1998.

[Anto 98b] G. Antoniol, R. Fiutem, and L. Cristoforetti. "Using metrics to identify design patterns in object-oriented software". In : *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International*, pp. 23–34, 1998.

[Aris 06] E. Arisholm. "Empirical assessment of the impact of structural properties on the changeability of object-oriented software.". *Information and Software Technology*, Vol. 48, No. 11, pp. 1046–1055, 2006.

[Aver 07] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta. "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*, pp. 385–394, ACM, New York, NY, USA, 2007.

[Ayal 13] Y. Ayalew and K. Mguni. "An Assessment of Changeability of Open Source Software.". *Computer and Information Science*, Vol. 6, No. 3, pp. 68–79, 2013.

[Badr 10] L. Badri, M. Badri, and F. Toure. "Exploring Empirically the Relationship between Lack of Cohesion and Testability in Object-Oriented Systems". In : T.-h. Kim, H.-K. Kim, M. Khan, A. Kiumi, W.-c. Fang, and D. Ślęzak, Eds., *Advances in Software Engineering*, pp. 78–92, Springer Berlin Heidelberg, 2010.

[Badr 11] L. Badri, M. Badri, and F. Toure. "An empirical analysis of lack of cohesion metrics for predicting testability of classes". *International Journal of Software Engineering and Its Applications*, Vol. 5, No. 2, pp. 69–85, 2011.

[Bans 02] J. Bansiya and C. Davis. "A hierarchical model for object-oriented design quality assessment". *Software Engineering, IEEE Transactions on*, Vol. 28, No. 1, pp. 4–17, Jan 2002.

[Basi 96] V. R. Basili, L. C. Briand, and W. L. Melo. "A validation of object-oriented design metrics as quality indicators". *IEEE Transactions on software engineering*, Vol. 22, No. 10, pp. 751–761, 1996.

[Beck 96] K. Beck, R. Crocker, G. Meszaros, J. Vlissides, J. O. Coplien, L. Dominick, and F. Paulisch. "Industrial Experience with Design Patterns". In : *Proceedings of the 18th International Conference on Software Engineering*, pp. 103–114, IEEE Computer Society, Washington, DC, USA, 1996.

[Bind 94] R. V. Binder. "Design for testability in object-oriented systems". *Communications of the ACM*, Vol. 37, No. 9, pp. 87–101, 1994.

[Brun 04] M. Bruntink and A. van Deursen. "Predicting class testability using object-oriented metrics". In : *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pp. 136–145, Sept 2004.

[Brun 06] M. Bruntink. "An empirical study into class testability". *Journal of Systems and Software*, Vol. 79, No. 9, pp. 1219 – 1232, 2006. Selected papers from the fourth Source Code Analysis and Manipulation (SCAM 2004) WorkshopFourth Source Code Analysis and Manipulation Workshop.

[Cald 94] V. Caldiera and H. D. Rombach. "The goal question metric approach". *Encyclopedia of software engineering*, Vol. 2, No. 1994, pp. 528–532, 1994.

[Chha 14] J. Chhabra and A. Parashar. "Prediction of changeability for object oriented classes and packages by mining change history". In : *Electrical and Computer Engineering (CCECE), 2014 IEEE 27th Canadian Conference on*, pp. 1–6, May 2014.

[Chid 94] S. R. Chidamber and C. F. Kemerer. "A Metrics Suite for Object Oriented Design". *IEEE Trans. Softw. Eng.*, Vol. 20, No. 6, pp. 476–493, June 1994.

[Chow 09] V. Chowdhary. "Practicing Testability in the Real World". In : *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*, pp. 260–268, April 2009.

[Comm 16] A. Commons. "The Apache Commons Mathematics Library". `http://commons.apache.org/proper/commons-math/`, 2016.

[Dofa 15] Dofactory. "Dofactory". 2015.

[Fere 05] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele. "Design pattern mining enhanced by machine learning". In : *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pp. 295–304, Sept 2005.

[Flur 07] B. Fluri. "Assessing Changeability by Investigating the Propagation of Change Types". In : *Software Engineering - Companion, 2007. ICSE 2007 Companion. 29th International Conference on*, pp. 97–98, May 2007.

[Gamm 95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gueh 08] Y.-G. Gueheneuc and G. Antoniol. "DeMIMA : A Multilayered Approach for Design Pattern Identification". *IEEE Transactions on Software Engineering*, Vol. 34, No. 5, pp. 667–684, 2008.

[Gueh 09] Y.-G. Guéhéneuc, H. Sahraoui, and et al. "Improving design-pattern identification : a new approach and an exploratory study". *Software Quality Journal*, 2009.

[Hege 12] P. Hegedus, D. Ban, R. Ferenc, and T. Gyimothy. "Myth or Reality ? Analyzing the Effect of Design Patterns on Software Maintainability". In : *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity*, pp. 138–145, Springer, 2012.

[Heit 07] I. Heitlager, T. Kuipers, and J. Visser. "A Practical Model for Measuring Maintainability". In : *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pp. 30–39, Sept 2007.

[Hust 01] B. Huston. "The effects of design pattern application on metric scores". *Journal of Systems and Software*, Vol. 58, No. 3, pp. 261 – 269, 2001.

[IEEE 98] IEEE. "IEEE Standard for Software Maintenance". *IEEE Std 1219-1998*, pp. i–, 1998.

[ISO 01] ISO 9126-1. "ISO/IEC 9126-1 :2001, Software engineering – Product quality – Part 1 : Quality model". Tech. Rep., International Organization for Standardization, 2001.

[Jure 10] M. Jureczko and D. Spinellis. *Using Object-Oriented Design Metrics to Predict Software Defects*, pp. 69–81. Vol. Models and Methodology of System Dependability of *Monographs of System*

Dependability, Oficyna Wydawnicza Politechniki Wroclawskiej, Wroclaw, Poland, 2010.

[Kaba 01] H. Kabaili, R. Keller, and F. Lustman. "Class Cohesion as predictor of changeability : An Empirical Study". 2001.

[Kell 99] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé. "Pattern-based Reverse-engineering of Design Components". In : *Proceedings of the 21st International Conference on Software Engineering*, pp. 226–235, ACM, New York, NY, USA, 1999.

[Khom 08] F. Khomh, Y.-G. Guéhéneuc, and P. Team. "An empirical study of design patterns and software quality". *GEODES–Research Group on Open, Distributed Systems, Experimental Software Engineering, University of Montreal*, 2008.

[Kris 97] M. S. Krishnan. *Cost and Quality Considerations in Software Product Management.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1997. AAI9802548.

[Lanz 07] M. Lanza and R. Marinescu. *Object-oriented metrics in practice : using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer Science & Business Media, 2007.

[Larm 05] C. Larman, E. Burr-Campillo, M.-C. Baland, and L. Carité. *UML 2 et les Design Patterns.* Pearson Education, 2005.

[Larm 12] C. Larman. *Applying UML and Patterns : An Introduction to Object Oriented Analysis and Design and Interative Development.* Pearson Education India, 2012.

[Lee 00] R. C. Lee and W. M. Tepfenhart. *UML and C++ : A Practical Guide to Object-Oriented Development.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd Ed., 2000.

[Lore 94] M. Lorenz and J. Kidd. *Object-oriented Software Metrics : A Practical Guide.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[Loza 08] A. Lozano and M. Wermelinger. "Assessing the effect of clones on changeability". In : *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pp. 227–236, Sept 2008.

[McGr 96] J. D. McGregor and S. Srinivas. "A measure of testing effort". In : *Proceedings of the 2nd conference on USENIX Conference on Object-Oriented Technologies (COOTS)-Volume 2*, pp. 10–10, USENIX Association, 1996.

[Prec 01] L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, and L. G. Votta. "A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions". *IEEE Trans. Softw. Eng.*, Vol. 27, No. 12, pp. 1134–1144, Dec. 2001.

[Pres 05] R. S. Pressman. *Software engineering : a practitioner's approach.* Palgrave Macmillan, 2005.

[Rosk 14] Z. Roško. "Predicting the Changeability of Software Product Lines for Business Application". In : *23rd International Conference on Information Systems Development*, Hrvatska znanstvena bibliografija i MZOS-Svibor, 2014.

[Sara 13] J. Saraiva, S. Soares, and F. Castor. "Towards a catalog of Object-Oriented Software Maintainability metrics". In : *Emerging Trends in Software Metrics (WETSoM), 2013 4th International Workshop on*, pp. 84–87, May 2013.

[Shir 03] J. S. Shirabad, T. C. Lethbridge, and S. Matwin. "Mining the maintenance history of a legacy software system". In : *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pp. 95–104, Sept 2003.

[Sing 10] Y. Singh and A. Saha. "Predicting testability of eclipse : a case study". *Journal of Software Engineering*, Vol. 4, No. 2, pp. 122–136, 2010.

[Slau 98] S. A. Slaughter, D. E. Harter, and M. S. Krishnan. "Evaluating the Cost of Software Quality". *Commun. ACM*, Vol. 41, No. 8, pp. 67–73, Aug. 1998.

[Soli 99] R. van Solingen and E. Berghout. *The Goal/Question/Metric Method : A Practical Guide for Quality Improvement of Software Development*. McGraw-Hill, 1999.

[Somm 01] I. Sommerville. *Software Engineering (6th Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Sun 12] X. Sun, B. Li, and Q. Zhang. "A Change Proposal Driven Approach for Changeability Assessment Using FCA-Based Impact Analysis.". In : X. Bai, F. Belli, E. Bertino, C. K. Chang, A. Elçi, C. C. Seceleanu, H. Xie, and M. Zulkernine, Eds., *COMPSAC*, pp. 328–333, IEEE Computer Society, 2012.

[Tahv 03] L. Tahvildari and K. Kontogiannis. "A metric-based approach to enhance design quality through meta-pattern transformations". In : *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, pp. 183–192, March 2003.

[Tsan 06] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis. "Design Pattern Detection Using Similarity Scoring". *Software Engineering, IEEE Transactions on*, Vol. 32, No. 11, pp. 896–909, Nov 2006.

[Voka 04a] M. Vokac. "Defect frequency and design patterns : an empirical study of industrial code". *Software Engineering, IEEE Transactions on*, Vol. 30, No. 12, pp. 904–917, Dec 2004.

[Voka 04b] M. Vokáč, W. Tichy, D. I. K. Sjøberg, E. Arisholm, and M. Aldrin. "A Controlled Experiment Comparing the Maintainability of Programs Designed with and Without Design Patterns&Mdash;A Replication in a Real Programming Environment". *Empirical Softw. Engg.*, Vol. 9, No. 3, pp. 149–195, Sep. 2004.