ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA
MAITRISE AVEC MÉMOIRE EN GENIE LOGICIEL
M. Sc. A.

PAR
Nesrine ABDELKAFI

ÉTUDE EMPIRIQUE SUR L'UTILISATION DES PATRONS JEE ET LEUR IMPACT SUR LA MODIFIABILITÉ DES APPLICATIONS JEE

MONTRÉAL, LE 23 JUIN 2015

PRÉSENTATION DU JURY

MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE :

Mme Ghizlane El Boussaidi, directeur de mémoire
Département de génie logiciel et TI à l'École de technologie supérieure

M. Yann-Gaël Guéhéneuc, codirecteur de mémoire
Département de génie informatique et génie logiciel à l'École Polytechnique de Montréal

M. Abdelouahed Gherbi, président du jury
Département de génie logiciel et TI à l'École de technologie supérieure

M. Roger Champagne, membre du jury
Département de génie logiciel et TI à l'École de technologie supérieure

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 23 JUIN 2015

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Je dédie ce mémoire à toutes les personnes qui ont contribuées de près ou de loin à la réalisation de mon projet de recherche.

Je tiens à remercier Mme. Ghizlane El Boussaidi, ma directrice de mémoire, pour m'avoir donné l'opportunité de réaliser ce projet de recherche et pour son suivi continue tout au long de mon travail et ses conseils constructifs. Je remercie énormément M. Yann-Gaël Guéhéneuc, mon codirecteur de mémoire, pour son aide et ses conseils précieux ainsi que pour son encouragement qui m'ont motivé davantage. Je remercie également le groupe de recherche du laboratoire en architecture de systèmes informatiques (LASI) et principalement Adnane, Alvine, Maroua, Sana,… pour leur soutien et leurs conseils.

Je remercie énormément le Ministère de l'enseignement supérieur et de la recherche scientifique de la Tunisie de m'avoir donné l'opportunité de poursuivre mes études de cycles supérieurs au Canada en m'attribuant une bourse nationale d'étude en Amérique du Nord. Je remercie également la mission universitaire de Tunisie en Amérique du nord (MUTAN) représenté en la personne de M. Lotfi Hassine pour son soutien, sa disponibilité et sa collaboration tout au long de la période de maitrise au Canada.

Je remercie infiniment mon père Najib, ma mère Bahija, mon frère Hassen, son épouse Emna et ma belle nièce Baya pour leur patience et leurs encouragements. Un grand merci à mes sœurs Fatma, Dorra et Malak pour leur soutien, leur présence et pour être mon adorable famille à Montréal. Je tiens aussi à remercier ma grande famille en Tunisie Sana, Nabila, Donia, Rawiya,…. et mes amies Wijden, Zeyneb, Faiza, …. pour leur présence et leur soutien moral.

ÉTUDE EMPIRIQUE SUR L'UTILISATION DES PATRONS JEE ET LEUR IMPACT SUR LA MODIFIABILITÉ DES APPLICATIONS JEE

Nesrine ABDELKAFI

RÉSUMÉ

Les patrons de conception décrivent des pratiques qui permettent la conception de logiciels de qualité. Beaucoup de patrons de conception fournissent des solutions qui visent à promouvoir la maintenabilité des logiciels et, plus précisément, la modifiabilité en facilitant la mise en œuvre de futurs changements du logiciel, ce qui réduit considérablement les coûts de la maintenance.

Plusieurs travaux visent à supporter l'utilisation des patrons de conception ou à identifier des occurrences de ces patrons dans des systèmes existants. Cependant, peu d'études empiriques évaluent l'impact des patrons JEE sur la modifiabilité des applications.

Dans ce mémoire, nous présentons une étude empirique sur l'utilisation des patrons JEE et leur impact sur la modifiabilité des applications JEE. Nous avons analysé 17 applications JEE libres pour identifier les patrons JEE qui sont utilisés et qui supportent la modifiabilité. Puis, nous avons évalué leur impact sur la modifiabilité de ces applications en étudiant les corrélations possibles entre l'utilisation de ces patrons et un ensemble de métriques liées à la modifiabilité. De plus, nous avons analysé différentes versions de trois de ces applications JEE afin d'évaluer l'évolution de la distribution des patrons appliqués et aussi l'impact de cette évolution sur la modifiabilité de ces applications. Afin de combler le manque d'outils qui permettent de détecter les patrons JEE, nous avons adapté l'outil Ptidej et nous avons développé un parseur basé sur l'arbre syntaxique du code source analysé. Une analyse manuelle des applications est effectuée pour valider les occurrences des patrons identifiés.

Les résultats de notre étude confirment qu'un bon nombre de patrons JEE supportant la modifiabilité sont utilisés dans les applications JEE. Ces patrons sont généralement ceux qui reflètent les bonnes pratiques d'une architecture en couches. Nous avons aussi constaté qu'il y a très peu de corrélations entre l'utilisation de ces patrons et la modifiabilité des applications. Les quelques corrélations observés suggèrent que l'application de certains patrons complexifient l'application. Cependant, notre analyse manuelle des applications JEE étudiées a permis de constater que l'utilisation de ces patrons facilite la compréhension du code et la délimitation des couches de l'architecture de ces applications.

Mots-clés: Patrons JEE, maintenance des applications, modifiabilité des logiciels, métriques.

AN EMPIRICAL STUDY OF THE USE OF JEE PATTERNS AND THEIR IMPACT ON THE MODIFIABILITY OF JEE APPLICATIONS

Nesrine ABDELKAFI

ABSTRACT

Software design patterns codify practices that support the design of quality software. Many design patterns provide solutions that claim to promote software maintenability and, specifically, modifiability as they ease the implementation of future changes to the software, which greatly reduces the maintenance costs.

Many researchers have been working on providing support for applying design patterns or identifying occurrences of these patterns. However, there is little empirical evidence that design patterns really improve modifiability.

In this work, we describe a preliminary study that investigates the use of JEE patterns and their impact on the modifiability of JEE applications. We analyze 17 open-source JEE applications to identify the JEE patterns that are used in these applications and that support their modifiability. Then, we evaluate the impact of these patterns on the modifiability of these applications by studying the correlations between the use of these patterns and a set of metrics related to software modifiability. We also analyzed different available versions of three analyzed JEE applications to assess the evolution of the distribution of applied patterns through these versions and also to assess the impact of this evolution on the modifiability of these applications. To detect occurrences of JEE patterns, we adapted the Ptidej tool and we developed a new parser that is based on the abstract syntax tree (AST) built from the analyzed source code. We performed a manual analysis of applications to validate the detected occurrences of patterns.

The results of our study confirm that many JEE patterns supporting modifiability are used in JEE applications. These patterns are usually those that reflect the best practices of a layered architecture. We also noticed that there is a very little correlation between the use of these patterns and the modifiability of applications. The few observed correlations suggest that the implementation of some patterns complicate the application. However, our manual analysis of JEE applications reveals that the use of these patterns facilitates the understanding of the code and the delimitation of architecture layers of these applications.

Keywords: JEE patterns, Software maintenance, software modifiability, software metrics.

TABLE DES MATIÈRES

Page

INTRODUCTION 1

CHAPITRE 1 REVUE DE LA LITTÉRATURE 7
1.1 Attribut de qualité 7
1.1.1 Qu'est-ce qu'un attribut de qualité ? 7
1.1.2 Exemple d'attribut de qualité : modifiabilité 8
1.2 Architecture logicielle 9
1.2.1 Qu'est-ce qu'une architecture logicielle ? 9
1.2.2 Exemple d'architecture qui supporte la modificabilité : le style en couche 10
1.3 Tactiques architecturales 14
1.3.1 Qu'est-ce qu'une tactique ? 14
1.3.2 Exemple de tactiques : les tactiques de modifiabilité 14
1.4 Métriques 16
1.5 Java Entreprise Edition : JEE 17
1.5.1 Qu'est-ce que JEE? 17
1.5.2 Architecture de JEE 18
1.5.3 Patrons JEE 19
1.6 Études sur l'utilisation et l'impact des patrons de conception 23
1.6.1 Travaux basés sur l'analyse des classes et leurs rôles dans les patrons 23
1.6.2 Travaux portant sur l'évaluation de l'impact des patrons sur la qualité 25
1.6.3 Études systématiques analysant l'état de l'art sur l'utilisation et l'évaluation des patrons 30
1.7 Approches centrées sur les patrons JEE 33
1.8 Conclusion 39

CHAPITRE 2 DÉFINITION DE L'ÉTUDE EMPIRIQUE 41
2.1 Questions de recherche 41
2.2 Choix des applications JEE 42
2.3 Choix des patrons JEE 47
2.4 Choix des métriques 56
2.5 Conclusion 62

CHAPITRE 3 COLLECTE DES DONNÉES 63
3.1 Calcul des métriques 63
3.1.1 Processus de calcul 63
3.1.2 Résultats de calcul 65
3.2 Détection des patrons JEE 68
3.2.1 Processus de la détection des patrons JEE 68
3.2.2 Résultat de la détection 74
3.3 Conclusion 76

CHAPITRE 4 ANALYSE DES DONNÉES 79
4.1 Jusqu'à quel degré les patrons JEE qui supportent la modifiabilité sont-ils utilisés dans les applications JEE? 79
4.1.1 Fréquences d'utilisation des patrons 79
4.1.2 Cooccurrences des patrons dans les applications 82
4.1.3 Distribution des patrons par domaine 84
4.2 Quel est l'impact de l'application de ces patrons JEE sur la modifiabilité des applications ? 85
4.2.1 Choix du coefficient de corrélation 86
4.2.2 Normalisation des occurrences des patrons par la taille de l'application 88
4.2.3 Résultats du calcul de coefficient de Spearman 92
4.2.4 Analyse des corrélations par patron 94
4.3 Analyse des résultats pour différentes versions d'applications JEE 96
4.3.1 Application '*Joindesk*' 96
4.3.2 Application '*mvnForum*' 98
4.3.3 Application '*Java Pet Store*' 101
4.4 Conclusion 103

CHAPITRE 5 SYNTHÈSE DES RÉSULTATS ET LIMITES DE L'ÉTUDE 105
5.1 Synthèse des résultats 105
5.2 Limites de l'étude 107
5.2.1 Validité externe des résultats de l'étude 107
5.2.2 Validité interne de l'étude 107

CONCLUSION & TRAVAUX FUTURS 109

ANNEXE I CORRESPONDANCES ENTRE LES PATRONS JEE ET LES TACTIQUES DE MODIFIABILITÉ 111

ANNEXE II VALIDATION DE LA NON-NORMALITE DES MÉTRIQUES POUR LE CHOIX DE COÉFFICIENT DE CORRÉLATION 121

LISTE DE RÉFÉRENCES BIBLIOGRAPHIQUES 125

LISTE DES TABLEAUX

Page

Tableau 1.1 Correspondance entre les styles architecturaux et les tactiques de modifiabilité Tirée de Bass, Clements et Kazman (2012, p.240) 13

Tableau 1.2 Catalogue des patrons JEE 20

Tableau 1.3 Les observations retenues du travail de Di Penta *et al.* 24

Tableau 1.4 Impact de trois patrons de conception sur les attributs de qualité Adapté de Khomh et Guéhéneuc (2008, p.276) 27

Tableau 1.5 Impact des patrons de conception sur trois attributs de qualité Adapté de Khomh et Guéhéneuc (2008, p.277) 28

Tableau 1.6 Impact de l'utilisation des patrons de conception sur les attributs de qualité Adapté de Ali et Elish (2013, p.6) 32

Tableau 2.1 Liste des applications JEE et leurs caractéristiques 43

Tableau 2.2 Correspondances entre les patrons JEE de la couche présentation et les tactiques de modifiabilité 49

Tableau 2.3 Correspondances entre les patrons JEE de la couche métier et les tactiques de modifiabilité 50

Tableau 2.4 Correspondances entre les patrons JEE de la couche intégration et les tactiques de modifiabilité 51

Tableau 2.5 Liste des patrons JEE dont les caractéristiques nous permettent de les détecter 53

Tableau 2.6 Relations entre les catégories des métriques et les tactiques de modifiabilité 56

Tableau 2.7 Nombre de métriques de chaque catégorie identifié dans les sept articles étudiés 58

Tableau 2.8 Liste des métriques retenues dans la première sélection 59

Tableau 3.1 Valeurs des métriques des 17 applications analysées 66

Tableau 3.2 Valeurs des métriques de quatre versions de l'application '*Joindesk*' 67

Tableau 3.3 Valeurs des métriques de quatre versions de l'application '*mvnForum*'...67

Tableau 3.4 Valeurs des métriques de deux versions de l'application '*Java Pet Store*'68

Tableau 3.5 Différentes combinaisons possibles pour détecter des patrons JEE en cherchant des occurrences de Façade avec l'outil *Ptidej*.....71

Tableau 3.6 Éléments utilisés pour détecter les patrons JEE en utilisant le parseur développé.....73

Tableau 3.7 Nombres de patrons JEE détectés dans chaque application.....75

Tableau 3.8 Nombres de patrons JEE identifiés pour chaque version de l'application '*Joindesk*'.....76

Tableau 3.9 Nombres de patrons JEE identifiés pour chaque version de l'application '*mvnForum*'.....76

Tableau 3.10 Nombres de patrons JEE identifiés pour chaque version de l'application '*Java Pet Store*'.....76

Tableau 4.1 Résultat de la normalisation du nombre de classes NOC.....90

Tableau 4.2 Nombres de patrons JEE normalisés dans chaque application.....91

Tableau 4.3 Coefficient de corrélation de Spearman entre la fréquence des patrons JEE et les métriques.....93

Tableau 4.4 Les valeurs de Valeur-P.....93

Tableau 4.5 Nombres de patrons JEE identifiés pour chaque version de l'application '*Joindesk*'.....97

Tableau 4.6 Valeurs des métriques de quatre versions de l'application '*Joindesk*'.....97

Tableau 4.7 Nombres de patrons JEE identifiés pour chaque version de l'application '*mvnForum*'.....99

Tableau 4.8 Valeurs des métriques de quatre versions de l'application '*mvnForum*'...99

Tableau 4.9 Nombres de patrons JEE identifiés pour chaque version de l'application '*Java Pet Store*'.....101

Tableau 4.10 Valeurs des métriques de deux versions de l'application '*Java Pet Store*'.....102

LISTE DES FIGURES

Page

Figure 0.1 Phases de la méthodologie 3

Figure 1.1 Style en couches 10

Figure 1.2 Couches du style en couches d'un système d'information Tirée de Larman *et al.* (2005, p.210) 11

Figure 1.3 Architecture en couches du système POS Tirée de Larman *et al.* (2005, p.542) 12

Figure 1.4 Architecture JEE Tirée d'Oracle (2013) 18

Figure 1.5 Diagramme de classes du patron « *Data Access Object* (*DAO*) » Tirée de Alur et al (2003, p.463) 21

Figure 1.6 Modèle en couches qui utilise les patrons JEE Tirée de Gilart-Iglesias *et al.* (2005, p.354) 34

Figure 1.7 Modèle proposé par Hammouda et Koskimies Tirée de Hammouda et Koskimies (2002, p.252) 36

Figure 2.1 Diagramme de classes du patron « *Intercepting Filter* » Tirée d'Alur et al (2003, p.145) 52

Figure 2.2 Diagramme de classes du patron « *Session Facade* » Tirée d'Alur et al (2003, p.343) 53

Figure 3.1 Processus de calcul des métriques 64

Figure 3.2 Relations entre les patrons JEE retenus pour notre étude 69

Figure 3.3 Processus de détection des patrons JEE 69

Figure 3.4 Présentation du patron « Façade » 70

Figure 4.1 Pourcentage d'utilisation de dix patrons JEE dans 17 applications JEE 80

Figure 4.2 Nombre d'occurrences des patrons JEE détectés et leur réparation dans chaque application 82

Figure 4.3 Démarche suivie pour le choix du coefficient de corrélation 86

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

AST	Abstract Syntax Tree
ACAIC	Ancestor Class Attribute Import Coupling
ACMIC	Ancestor Class Method Import Coupling
ADIH	Average Depth of Inheritance Hierarchy
CA	Afferent Coupling
CBO	Coupling Between Objects
CE	Efferent Coupling
CHC	Changing Classes
CK	Chidamber & Kemerer
Ckjm	Chidamber & Kemerer Java Metrics
CLD	Class to Leaf Depth
CM	Changing Methods
CYCLO	Cyclomatic complexity
DAO	Data Access Object
DeMIMA	Design Motif Identification Multi Layered Approach
DIT	Depth of Inheritance Tree
DTD	Document Type Definition
EJB	Entreprise Java Beans
FR	Functional Requirements
GoF	Gang of Four
HTTP	Hypertext Transfer Protocol
IS	Information System

JASIG	Java in Administration Special Interest Group
JEE	Java Entreprise Edition
J2EE	Java 2 Platform Entreprise Edition
JSP	Java Server Pages
JVM	Java Virtual Machine
LASI	Laboratoire en Architecture de Systèmes Informatiques
LCOM	Lack of Cohesion of Methods
LCOM1	Lack of Cohesion in Method 1
LCOM2	Lack of Cohesion in Method 2
LOC	Lines of Code
MVC	Model View Controller
NA	Number of Attributes
NFR	Non Functional Requirements
NMA	Number of Methods Added
NMI	Number of Methods Inherited
NMO	Number of Methods Overriden
NOC	Number of Classes
NOD	Number of Descendant class
NOM	Number of Methods
NOP	Number of Paquetages
NOPa	Number of Parents
NT	New Technology

OLEX	OpenLoic EXchange
OSI	Open Systems Interconnection
POJO	Plain Old Java Object
POS	Point of Sale
Ptidej	Pattern Trace Identification, Detection and Enhancement in Java
PV	Program Vo
RAM	Random Access Memory
REST	Representational State Transfer
RFC	Response set For a Class
SIX	Specialization Index
SQL	Structured Query Language
UML	Unified Modeling Language
WMC	Weigthed Method Per Class
WMPC1	Weighted Methods Per Class 1
WMPC2	Weighted Methods Per Class 2
XML	eXtensible Markup Language

INTRODUCTION

Contexte et problématique

La conception logicielle est un processus difficile. Elle nécessite de l'expérience et des connaissances techniques mais aussi des connaissances du domaine pour lequel on développe une application logicielle.

Pour supporter les concepteurs dans ce processus, les chercheurs et les participants ont recensé un certain nombre de patrons de conception qui sont connus comme étant des solutions à des problèmes récurrentes. En effet, les patrons ont été proposés à différents niveaux du processus de conception : des styles architecturaux (Bass, Clements et Kazman, 2012), des patrons de conception (Gamma et al., 1994) et des patrons élémentaires telles que les tactiques (Bachmann, Bass et Nord, 2007).

Un style architectural décompose le système logiciel en un ensemble de modules de façon à promouvoir certains attributs de qualité. Par exemple, l'architecture en couches organise un système en une hiérarchie de couches ordonnées du plus bas vers le plus haut niveau d'abstraction. Chaque couche est constituée de modules qui font appel aux services des composants de la couche immédiatement inférieure (Buschmann et al., 1996).

Un patron de conception résout un problème de conception détaillée et, de ce fait, a une granularité plus fine qu'un style architectural. Par exemple, dans une conception orientée objet, le patron « *Singleton* » permet de s'assurer d'avoir une seule occurrence d'une classe dont la visibilité est globale. L'ensemble des patrons de conception proposent des solutions à des problèmes récurrents. Ils ont été généralement décrits de façon semi-formelle, c'est-à-dire, la description d'un patron est faite de façon textuelle accompagnée de diagrammes illustrent la forme de la solution proposée.

Plusieurs approches ont été proposées pour formaliser la solution proposée par le patron soit pour : (i) supporter le concepteur dans l'application du patron ou (ii) détecter des occurrences du patron dans un logiciel analysé.

Cependant, très peu de travaux se sont penchés sur l'étude du degré d'utilisation des patrons et l'évaluation de leur impact réel sur la qualité du logiciel. En effet, pour que les patrons soient un mécanisme efficace pour véhiculer l'expérience en conception, il faut s'assurer qu'ils sont utilisables par des concepteurs et que la conception obtenue par cette utilisation est optimale (Zhang et Budgen, 2012). Dans ce contexte, il est nécessaire de réaliser des études empiriques visant à évaluer l'utilisation des patrons, d'une part, et leur impact sur les attributs de qualité, d'autre part.

Objectifs

Le but général de ce projet de recherche est d'étudier l'utilisation des patrons de conception et d'évaluer l'impact de ces patrons sur la qualité des applications.

Dans ce projet, nous limitons notre analyse aux applications ayant une architecture 'multi-niveaux', architecture communément utilisée dans le développement de logiciels d'envergure. Une architecture 'multi-niveaux' est une architecture composé de plusieurs niveaux (e.g: l'architecture client-serveur). Dans le reste du document, nous ne différencions pas la signification des deux termes 'architecture en couches' et 'architecture multi-niveaux'. En particulier, nous analysons des applications JEE[1] qui sont des exemples d'applications en couches et qui sont très rarement analysées dans la littérature. La mise en œuvre de l'architecture en couches nécessite l'application de plusieurs patrons de conceptions. Ces patrons peuvent avoir un impact sur les attributs de qualité supportés par cette architecture (e.g: portabilité, testabilité, modifiabilité). Nous évaluons l'impact de ces patrons sur un seul

[1] Applications développées en utilisant le Framework Java Entreprise Edition (JEE).

attribut de qualité, à savoir la modifiabilité. Parce que nous nous concentrons sur les applications JEE, nous nous intéressons aux patrons de conception JEE.

À cet effet, les objectifs spécifiques de ce projet de maîtrise sont :

- identifier les patrons JEE[2] qui supportent la modifiabilité;
- analyser le degré d'utilisation de ces patrons dans les applications JEE;
- évaluer l'impact de ces patrons sur la modifiabilité des applications.

Méthodologie de recherche

Pour atteindre notre objectif, nous avons suivi la méthodologie illustrée par la Figure 0.1.

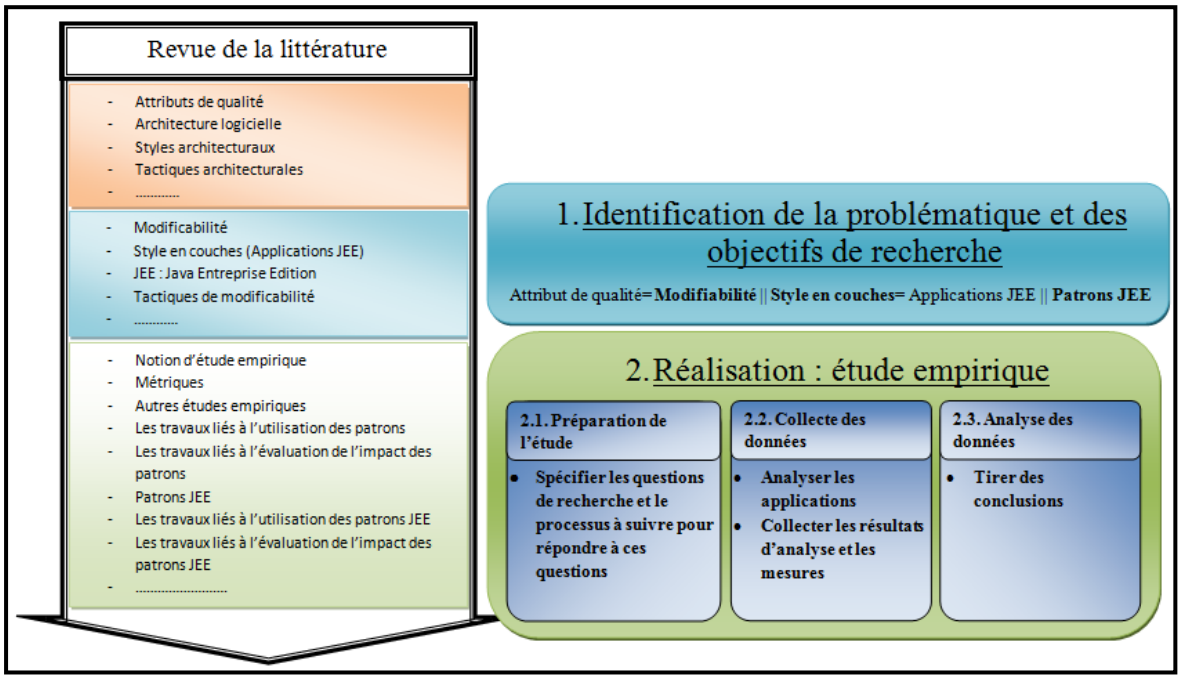

Figure 0.1 Phases de la méthodologie

[2] Les patrons JEE présentent des solutions à des problèmes récurrents identifiés lors de développement des applications sur la plateforme JEE.

D'abord, nous présentons l'état de l'art que nous avons effectué sur notre projet. Cet état de l'art, qui présente une étape d'initiation de notre travail, se focalise sur des notions de base relatives à notre projet. Il représente une sous-phase de la phase de 'Revue de la littérature' qui s'étale tout au long du projet. La première phase vise à identifier principalement les objectifs de notre projet de recherche. Pour ce faire, nous avons réalisé une revue préliminaire de littérature qui survole des concepts de base tels que les styles architecturaux, les patrons de conception, les attributs de qualité et quelques approches portant sur l'utilisation des patrons.

Ensuite, nous réalisons une étude empirique dans la deuxième phase de notre méthodologie. Cette étude a pour objectif d'identifier les patrons JEE utilisés dans les applications JEE et d'évaluer leurs impacts sur la modifiabilité de ces applications. La notion d'étude empirique est définie par Wohlin *et al.* (2012), comme étant une étude de cas qui s'appuie sur plusieurs sources de preuves afin d'examiner un phénomène de génie logiciel.

Conformément à la description fournie par Wohlin *et al.* et à d'autres études faites dans le domaine de génie logiciel (Bieman et al., 2003), (Correia, Kanellopoulos et Visser, 2009), (Fontana et al., 2013), (Ng et al., 2006), notre étude empirique se compose de trois étapes. Ces différentes étapes sont illustrées dans la Figure 0.1. L'étape 2.1 consiste à spécifier (i) les questions de recherche auxquelles l'étude vise à répondre ainsi que (ii) le processus à suivre pour répondre à ces questions. Notre processus inclut le choix d'un ensemble de patrons JEE à étudier, d'un ensemble d'applications JEE à analyser et d'un ensemble de métriques pour évaluer la modifiabilité. L'étape 2.2 consiste à analyser les applications et collecter les résultats d'analyse (les valeurs des métriques et les nombres de patrons détectés). Une fois les données collectées, il s'agit enfin à l'étape 2.3 d'analyser ces données afin de répondre aux questions précisées dans l'étape 2.1 et de tirer des conclusions.

Organisation du mémoire

Ce mémoire est organisé comme suit : le chapitre 1 présente un ensemble de définitions de base ainsi qu'un état de l'art sur les travaux liés à l'utilisation des patrons et à l'évaluation de

leur impact sur la qualité des systèmes. Notre étude empirique est décrite dans les trois chapitres qui suivent. Dans le chapitre 2, nous détaillons la première étape de l'étude empirique en présentant les questions de recherche auxquelles l'étude essaye de répondre ainsi que les choix à faire pour répondre à ces questions. La collecte des données, incluant le calcul des métriques de modifiabilité et la détection des patrons JEE, est présentée dans le chapitre 3. La discussion et l'interprétation des résultats sont présentées dans le chapitre 4. La synthèse des résultats et les limites de notre étude sont décrites dans le chapitre 5. La conclusion générale ainsi que les perspectives du projet sont présentées dans le dernier chapitre.

CHAPITRE 1

REVUE DE LA LITTÉRATURE

Nous commençons ce chapitre par la définition d'un ensemble de concepts reliés à notre projet de recherche. Ces concepts comprennent les notions d'attribut de qualité, d'architecture logicielle, de tactiques architecturales, de métriques et de *Java Entreprise Edition* (JEE). Ensuite, nous présentons un survol sur les travaux liés à l'utilisation des patrons et à l'évaluation de leur impact sur la qualité des systèmes en distinguant entre les travaux qui portent sur les patrons de conception de ceux qui traitent les patrons JEE.

1.1 Attribut de qualité

1.1.1 Qu'est-ce qu'un attribut de qualité ?

Les exigences logicielles sont connues comme étant les besoins et les contraintes qui doivent être pris en considération lors du développement des systèmes. De façon générale, les exigences logicielles sont divisées en en deux catégories : les exigences fonctionnelles (FRs) et les exigences non fonctionnelles (NFRs) (Dean Leffingwell, 2003) et (Abran, 2004). Les exigences fonctionnelles sont définies comme étant les fonctionnalités du système alors que les exigences non fonctionnelles représentent les contraintes que le système doit satisfaire.

Dans (Bass, Clements et Kazman, 2012), les exigences logicielles sont divisées en trois catégories : exigences fonctionnelles, contraintes et exigences d'attributs de qualités. Un attribut de qualité est défini comme une « propriété mesurable ou testable du système utilisée pour indiquer à quel degré le système satisfait les besoins des parties prenantes » (Bass, Clements et Kazman, 2012, p.63).

1.1.2 Exemple d'attribut de qualité : modifiabilité

La nomenclature des attributs de qualité diffère dans la littérature. Dans ce rapport, l'appellation que nous allons utiliser est celle de (Bass, Clements et Kazman, 2012).

Selon (Bass, Clements et Kazman, 2012), « la modifiabilité est un attribut de qualité dont l'intérêt se focalise sur le coût et le risque de faire des modifications » (Bass, Clements et Kazman, 2012, p.117). De même, (Bachmann, Bass et Nord, 2007) présentent la modifiabilité comme étant « un attribut de qualité de l'architecture logicielle qui concerne le coût du changement et fait référence à la facilité avec laquelle un système logiciel peut s'adapter aux changements ». (Bachmann, Bass et Nord, 2007, p.7)

Il est important de distinguer entre la modifiablité et la notion de la prédisposition aux changements (« *Change-proneness* »). Cette dernière fait référence à des éléments d'un système qui sont sujets à changement et dans ce cas « cela indique les problèmes sous-jacents de qualité » (Bieman et al., 2003, p.44).

Pour évaluer chaque changement effectué, l'architecte doit répondre à ces quatre questions (Bass, Clements et Kazman, 2012), (Bachmann, Bass et Nord, 2007) :

(1) Qui fait le changement ?
(2) Quand peut-on faire le changement ?
(3) Qu'est-ce qui peut changer ?
(4) Quel est le coût du changement ?

Pour satisfaire les attributs de qualité, on utilise des styles, des patrons architecturaux ainsi que des patrons élémentaires de conception (c.-à-d. des tactiques). Quant à l'évaluation de la qualité des systèmes (c.-à-d. l'évaluation de la satisfaction des attributs de qualité), on se sert d'un ensemble de mesures (c.-à-d. des métriques). Nous présentons dans les sections suivantes les notions d'architecture logicielle (section 1.2), de tactiques (section 1.3) et de métriques (section 1.4).

1.2 Architecture logicielle

1.2.1 Qu'est-ce qu'une architecture logicielle ?

L'architecture logicielle est définie comme étant « l'ensemble des structures requises pour raisonner à propos du système, ce qui comprend des éléments logiciels, des relations entre eux et des propriétés de ces éléments et ces relations» (Bass, Clements et Kazman, 2012, p.4). Aussi, (Kazman, Bass et Klein, 2006) présentent l'architecture logicielle comme étant un artefact défini en termes d'éléments dont la granularité est grossière. Plus précisément, selon (Kazman, Bass et Klein, 2006) le but de représenter l'architecture logicielle d'un système est (i) de spécifier la conception de systèmes sous une forme compréhensible qui va servir par la suite comme support de communication et (ii) de détailler la spécification du système pour s'assurer de la satisfaction de toutes les exigences. De façon simple, l'architecture logicielle présente les éléments du système, leurs relations ainsi que leurs interactions. Un élément peut être un module (une unité d'implémentation) ou un composant (une entité d'exécution). Dans le reste du document, nous allons utiliser le terme module car nous nous intéressons à l'architecture en couches qui est une vue du système basée sur les modules (Clements et al., 2010).

Plusieurs styles d'architectures ont été recensés dans la littérature. En effet, une architecture est construite en utilisant un style (ou plusieurs) architectural. (Buschmann et al., 1996) définissent le style architectural comme étant « un problème de conception récurrent et particulier qui se pose dans des contextes spécifiques de conception » (Buschmann et al., 1996, p.394). Tandis que selon (Garlan et Shaw, 1993), le style architectural est « une spécialisation de types d'éléments et de relations, avec un ensemble de contraintes sur la façon dont ils peuvent être utilisés » (Garlan et Shaw, 1993, p.6). Des exemples de styles architecturaux sont : le style en couches, le style « *pipe and filter* » et le style orienté service. Chacun de ces styles supporte un ensemble spécifique d'attributs de qualité.

1.2.2 Exemple d'architecture qui supporte la modificabilité : le style en couche

Le style en couches est l'un des styles les plus connus dans la littérature. Plusieurs travaux décrivent ce style en détail, tels que (Buschmann et al., 1996), (Bachmann, Bass et Nord, 2007) et (Garlan et Shaw, 1993). Selon (Bachmann, Bass et Nord, 2007), le style en couches permet de « structurer des applications en les décomposant en groupes de sous-tâches dans lequel chaque groupe de sous-tâches correspond à un niveau particulier d'abstraction » (Bachmann, Bass et Nord, 2007, p.25). Ce style appartient à la famille de modèles '*From Mud to Structure*' qui a été décrit par (Buschmann et al., 1996) comme étant les patrons qui décomposent une tâche complexe en plusieurs sous-tâches.

Buschmann *et al.* (1996) présentent la structure résultante de l'application de ce style comme étant une structure hiérarchique composée d'un nombre approprié de couches ordonnées du plus bas vers le plus haut niveau d'abstraction, dont chaque couche est constituée de composants qui font appel aux services des composants de la couche immédiatement inférieure. La Figure 1.1 illustre la structure du style en couches. Comme illustrée par cette figure, la couche J joue un double rôle, « elle fournit les services qui vont être utilisés par la couche J +1 et elle collabore avec la couche J-1 en déléguant des sous-tâches à cette dernière » (Buschmann et al., 1996, p.34).

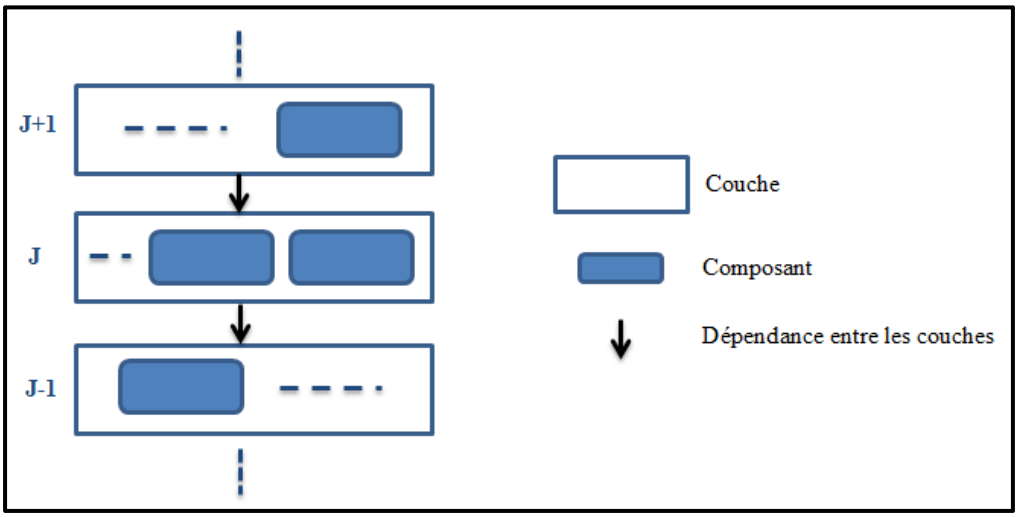

Figure 1.1 Style en couches

La Figure 1.2 illustre un exemple d'un ensemble de couches.

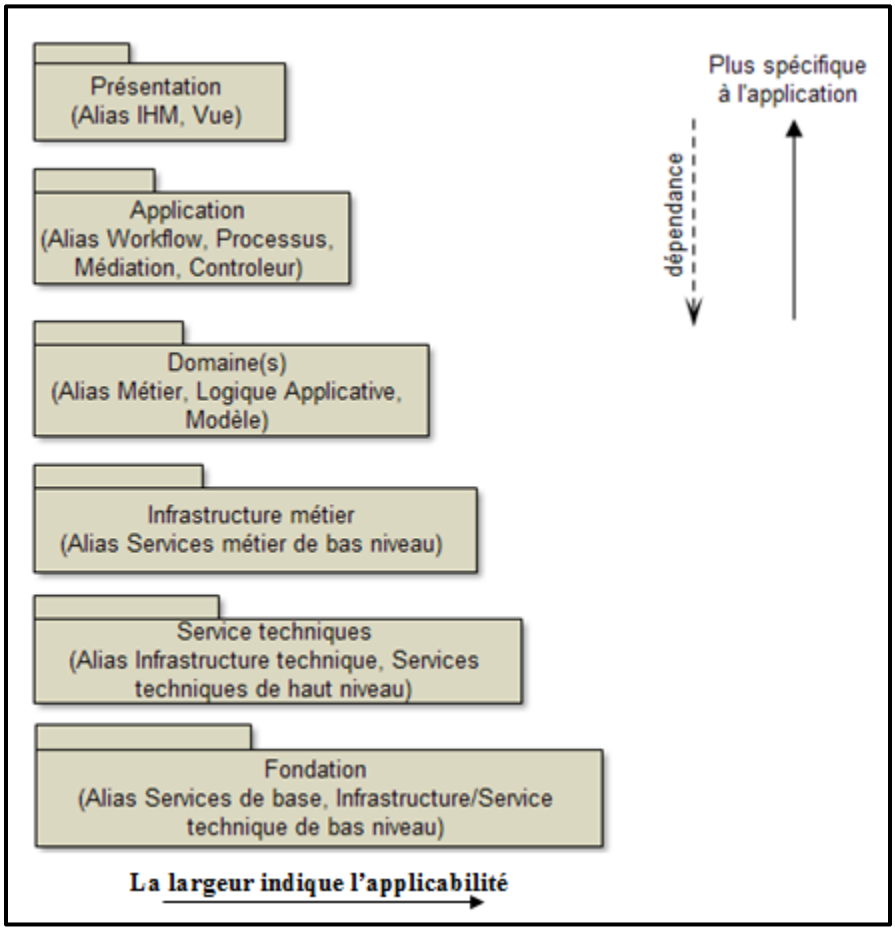

Figure 1.2 Couches du style en couches
d'un système d'information
Tirée de Larman *et al.* (2005, p.210)

Plusieurs systèmes connus utilisent le style en couches, citons la machine virtuelle Java (JVM), les systèmes d'information (IS), Windows nouvelle technologie (NT) et le modèle Open Systems Interconnection (OSI) (Buschmann et al., 1996).

À titre indicatif, les différentes couches présentées dans la Figure 1.2 ne figurent pas obligatoirement dans toutes les décompositions. Le nombre de couches dépend de plusieurs facteurs dont la taille, la complexité et le type du système.

Par exemple, la Figure 1.3 présente l'architecture en couches d'un système de gestion de ventes (Point of Sale (POS)) qui a été introduit dans (Larman et al., 2005). Ce système est décrit comme étant « une application informatisée qui sert notamment à l'enregistrement des ventes et au traitement des règlements. Il est constitué de composants matériels (ordinateurs et lecteurs de codes à barres) et d'un logiciel. Il est connecté à diverses applications telles que des systèmes de calcul des taxes et de gestion des stocks » (Larman et al., 2005, p.51).

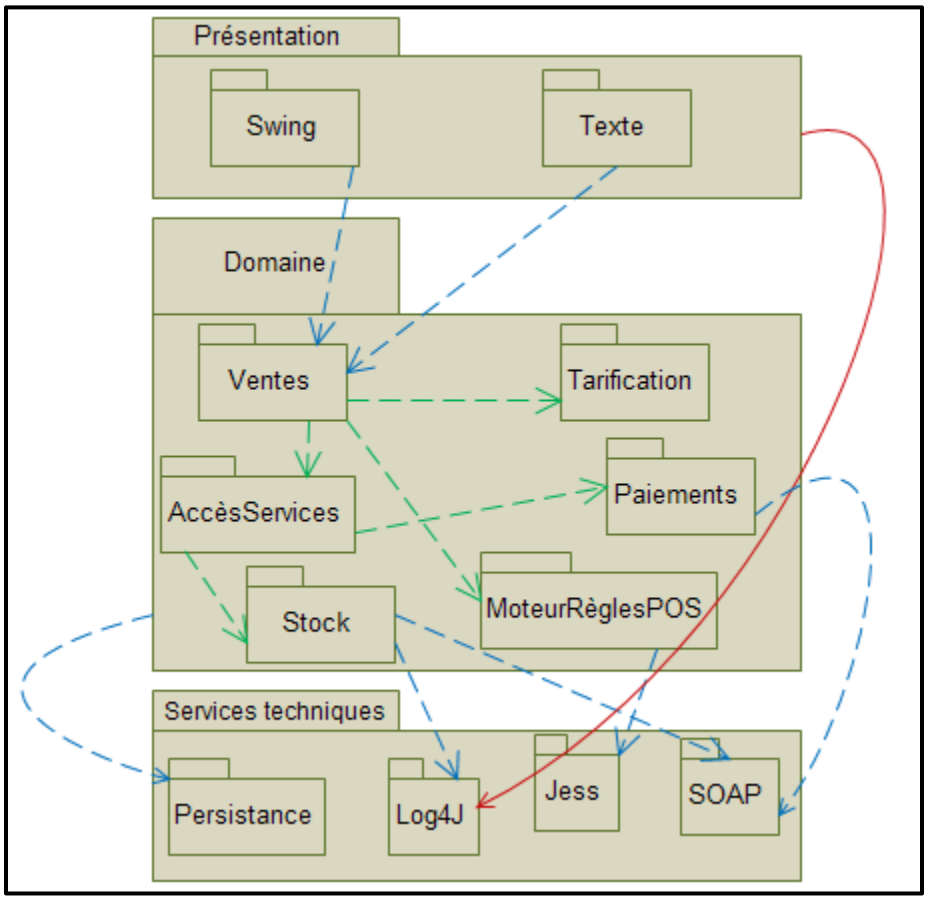

Figure 1.3 Architecture en couches du système POS
Tirée de Larman *et al.* (2005, p.542)

Comme illustrée par la Figure 1.3, l'architecture du système POS est composé de trois couches: (1) la couche présentation qui contient les paquetages des interfaces graphiques, (2) la couche domaine qui regroupe les paquetages liés à la logique métier de l'application et (3) la couche service technique qui englobe les services techniques de bas niveau.

Selon Garlan et Shaw (1993) et Buschmann et al (1996), le style en couches facilite la maintenabilité (la capacité de l'application logicielle à être modifiée (Losavio et al., 2003)), la portabilité (la capacité de l'application logicielle à être transférée d'un environnement à un autre (Losavio et al., 2003)), la testabilité du système (la capacité de l'application logicielle à être validée (Losavio et al., 2003)) ainsi que la réutilisation des couches. De plus, (Bass, Clements et Kazman, 2012) ont établi explicitement la correspondance entre ce style et la modifiabilité en précisant les tactiques de modifiabilité qui peuvent être utilisés pour implémenter ce style. Le Tableau 1.1 présent des exemples de patrons qui supportent la modificabilité. La notion de tactique est discutée dans la section suivante.

Tableau 1.1 Correspondance entre les styles architecturaux et les tactiques de modifiabilité
Tirée de Bass, Clements et Kazman (2012, p.240)

	Modifiability										
	Increase Cohesion		Reduce Coupling					Defer Binding Time			
Pattern	Increase Semantic Coherence	Abstract Common Services	Encapsulate	Use a Wrapper	Restrict Comm. Paths	Use an Intermediary	Raise the Abstraction Level	Use Runtime Registration	Use Startup-Time Binding	Use Runtime Binding
Layered	X	X	X		X	X	X			
Pipes and Filters	X		X		X	X			X	
Blackboard	X	X			X	X	X	X		X
Broker	X	X	X		X	X	X	X		
Model View Controller	X		X			X				X
Presentation Abstraction Control	X		X			X	X			
Microkernel	X	X	X		X	X				
Reflection	X		X							

L'utilisation de ce patron influence négativement la performance des systèmes. Par ailleurs, l'inconvénient majeur de ce patron est la difficulté de subdiviser correctement des systèmes en une hiérarchie en couches ((Garlan et Shaw, 1993), (Buschmann et al., 1996)).

La mise en œuvre d'une architecture conformément à un style architectural nécessite l'utilisation de patrons de granularité plus fine comme les patrons de conception et les tactiques architecturales. Ces concepts sont abordés dans les sections suivantes.

1.3 Tactiques architecturales

1.3.1 Qu'est-ce qu'une tactique ?

Bass, Clements et Kazman (2012) définissent le terme tactique de façon générale comme étant « une décision de conception qui affecte l'atteinte d'une réponse souhaitée pour un attribut de qualité » (Bass, Clements et Kazman, 2012, p.70). Bass, Clements et Kazman (2003) présentent la tactique architecturale en tant qu'« un bloc de construction réutilisable qui fournit une solution architecturale construite en se basant sur l'expérience pour contribuer à la réalisation d'un attribut de qualité » (Bass, Clements et Kazman, 2003, p.100). Ces tactiques peuvent être considérées « comme des composantes de base à partir desquels les patrons et les styles architecturaux sont créés » (Bass, Clements et Kazman, 2003, p.125). D'une façon simple, nous présentons une tactique architecturale comme étant une transformation de point de vue architectural dont le but est de satisfaire un attribut de qualité.

1.3.2 Exemple de tactiques : les tactiques de modifiabilité

Dans la littérature, plusieurs travaux, tels que (Bass, Clements et Kazman, 2012) et (Bachmann, Bass et Nord, 2007), ont décrit les tactiques de modifiabilité.

Ces travaux classifient les tactiques de modifiabilité en quatre groupes selon leurs objectifs :

1. La réduction de la taille d'un module qui comporte une seule tactique « diviser le module *(Split Module)* ». Cette tactique consiste à diviser un module en plusieurs modules de taille inférieure afin de réduire le coût des changements.
2. L'augmentation de la cohésion qui comporte une seule tactique « augmenter la cohésion sémantique *(Increase semantic coherence) ».* Cette tactique vise à augmenter la cohésion d'un module en déplaçant les responsabilités qui ne

fournissent pas les mêmes services de ce module vers un autre (un module existant ou un nouveau module) pour minimiser l'impact des changements.

3. La réduction du couplage qui comporte cinq tactiques :
    - « Encapsuler (*Encapsulate)* » *:* cette tactique consiste à introduire une interface qui encapsule les responsabilités afin de limiter l'interaction directe avec le module et de réduire la propagation des changements.
    - « Limiter les dépendances *(Restrict dependencies) » :* cette tactique consiste à limiter l'accès aux modules autorisés uniquement. Elle est très utilisée dans les architectures en couches.
    - « Utiliser un intermédiaire *(Use an intermediary) »:* cette tactique consiste à éliminer la dépendance entre deux modules en introduisant un intermédiaire.
    - « Restructurer *(Refactor) » :* Cette tactique consiste à refactoriser le code source (dans le cas où le code source est dupliqué ou complexe) en localisant les responsabilités communes et en créant un module parent de ces sous modules refactorisés.
    - « Abstraire les services communs *(Abstract common services) »:* cette tactique consiste à regrouper les services similaires et les implémenter comme étant des services abstraits pour minimiser le coût de modification.
4. Le « *Defer Binding* ». Les tactiques de ce groupe consistent à concevoir des modules (ou artefacts) qui sont paramétrables et dont les valeurs des paramètres peuvent être fournies le plus tard possible dans le cycle de vie (i.e., à la compilation, à l'exécution, au déploiement, etc.). Ces tactiques visent à maximiser les changements effectués par l'ordinateur et minimiser l'intervention du développeur pour effectuer les modifications, par exemple le polymorphisme et les fichiers de configurations (*properties*) sont deux exemples de mécanismes qui supportent le '*Defer binding*'.

Nous nous appuierons dans les chapitres suivants sur la définition des tactiques de modifiabilité présentée dans cette section.

1.4 Métriques

Une métrique logicielle est une mesure quantifiable utilisée pour évaluer une certaine propriété d'un logiciel. Les métriques sont classifiées en plusieurs catégories selon les propriétés qu'elles mesurent, par exemple des métriques de taille, de complexité, de couplage, de cohésion, d'héritage, etc.

Plusieurs travaux ont proposé et analysé des métriques logicielles (Chidamber et Kemerer, 1994), (Bieman et Kang, 1995), (Henderson-Sellers, 1996) et (Briand, Daly et Wüst, 1998). Parmi les métriques logicielles orientées objet les plus connus et utilisées, on trouve les métriques de Chidamber & Kemerer (CK) (Chidamber et Kemerer, 1994). Ces derniers ont proposé six métriques qui couvrent quatre catégories différentes de métriques :

- Métriques de complexité : (1) *Weighted Method per Class* (WMC) qui additionne les complexités de toutes les méthodes d'une classe. La complexité d'une méthode peut être calculée en utilisant plusieurs méthodes (e.g: la complexité cyclomatique).
- Métriques d'héritage : (2) *Depth of Inheritance Tree* (DIT) qui représente la profondeur d'une classe dans une arborescence d'héritage et (3) *Number of Children* (NOC) qui compte le nombre de sous-classes d'une classe parente.
- Métriques de couplage : (4) *Coupling Between Objects* (CBO) qui compte le nombre de classes couplées à une classe spécifique et (5) *Response set For a Class* (RFC) qui calcule le nombre de méthodes (les méthodes de la classe étudiée ainsi que celles appelées par ces méthodes) qui peuvent être exécutées en répondant à une demande envoyée à une classe.
- Métrique de cohésion : (6) *Lack of Cohesion of Methods* (LCOM) qui soustrait le nombre de paires des méthodes dans une classe qui n'utilisent aucun attribut en commun du nombre de paires des méthodes dans la même classe qui utilisent au moins un attribut en commun.

Les métriques sont utilisées dans la littérature pour différents objectifs. Par exemple, elles peuvent être utilisées pour détecter les défauts de conception (Tahvildar et Kontogiannis, 2004), ou évaluer la qualité des logiciels (Tahvildar et Kontogiannis, 2004), (Fontana et al.,

2013). Par ailleurs, de nouvelles métriques sont apparues pour combler certains besoins. Par exemple, (Kakarontzas et al., 2013) proposent une nouvelle métrique qui permet de vérifier la compatibilité des systèmes orientés objet avec l'exigence de la réutilisation. Cette nouvelle métrique est basée sur les six métriques de Chidamber & Kemerer (CK) déjà décrites.

1.5 Java Entreprise Edition : JEE

Dans cette section, nous présentons le *Framework Java Entreprise Edition* (JEE), son architecture ainsi que les patrons JEE.

1.5.1 Qu'est-ce que JEE?

JEE est l'abréviation de « *Java Entreprise Edition »*, anciennement connue par le nom *Java 2 Platform, Enterprise Edition* (J2EE). JEE est une plate-forme pour développer, déployer et exécuter des applications distribuées. Cette plateforme fournit un ensemble de services techniques (e.g., la gestion de la sécurité, l'accès à la base de données, etc.) pour faciliter la réalisation de certaines tâches et permettre au développeur de se focaliser sur la logique métier de l'application (Jean-Michel, 1999). La plateforme JEE[3] est conçue pour aider les développeurs à créer des applications multi-niveaux, de grande échelle, extensibles, fiables et sécurisées.

Six versions de JEE existent dans la littérature. Chaque version est caractérisée par une liste de technologies qu'elle supporte. J2EE 1.2 (12 décembre 1999) est la première version de cette plateforme et la dernière version est JEE 7 (12 Juin 2013).

Le *Framework* JEE offre plusieurs composants qui facilitent le développement d'applications web : les composants clients (par exemple les *applet*s), les composants web (par exemple les

[3] Oracle. 2014. Java Documentation, « Java Platform, Enterprise Edition (Java EE) 7 ». En ligne. <https://docs.oracle.com/javaee/7/firstcup/index.html>. Consulté le 24 avril 2015.

servlets et les pages *Java Server Pages* (JSP)) et les composants métiers (par exemple les *Entreprises Java Beans* (EJBs)).

1.5.2 Architecture de JEE

Le *Framework* JEE est organisé selon une architecture multi-niveaux qui est composée en général de quatre niveaux (Alur et al., 2003), (Johnson, 2004), (Niziaek, Zabierowski et Napieralski, 2008). La Figure 1.4 illustre les différentes couches de l'architecture JEE.

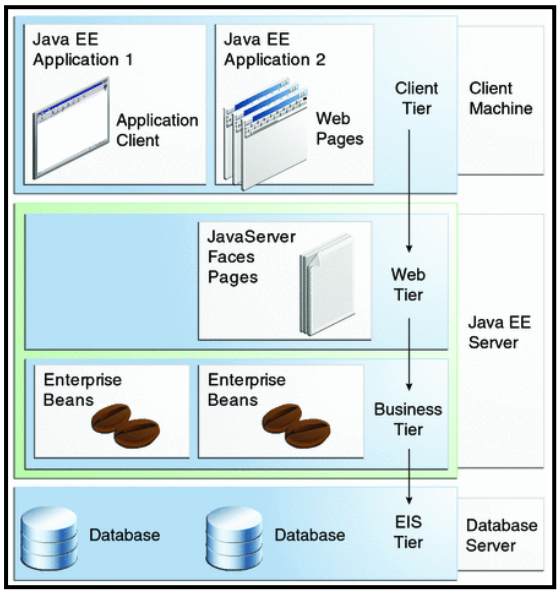

Figure 1.4 Architecture JEE
Tirée d'Oracle[4] (2013)

Les quatre niveaux qui décrivent l'architecture JEE sont :

(1) Le niveau client : offre une interface à travers laquelle le client interagit avec l'application.

(2) Le niveau web (appelée aussi la couche présentation) : gère les requêtes des utilisateurs.

[4] Oracle. 2013. « The Java EE 6 Tutorial». En ligne. <http://docs.oracle.com/javaee/6/tutorial/doc/bnaay.html>. Consulté le 24 avril 2015.

(3) Le niveau métier : implémente la logique métier de l'application.
(4) Le niveau intégration : est responsable de l'interaction avec les systèmes externes, par exemple une base de données.

1.5.3 Patrons JEE

La notion de patron de conception a été discutée dans plusieurs livres de référence (Gamma et al., 1994), (Horstmann, 2009) et (Bass, Clements et Kazman, 2012). Un patron de conception propose une solution à un problème récurrent. Bass, Clements et Kazman (2012) l'ont décrite comme étant une solution à une catégorie de problèmes dans un contexte général. La solution proposée est décrite par le triplet (contexte, problème et solution). Cette solution est utilisée dans un contexte spécifique pour résoudre un problème particulier. De plus, Chang et al. (2009) décrivent les patrons comme étant des solutions réutilisables pour résoudre des problèmes connus.

La notion de patron JEE est introduite par Alur et al dans leur livre « Core J2EE patterns: Best practices and design stratégies » en 2003. Alur *et al.* (2003) se sont basés sur leurs expériences pour décrire les patrons JEE qui sont définis comme étant les solutions aux problèmes récurrents identifiés lors de développement des applications sur la plateforme JEE. Ils présentent 21 patrons JEE. Comme les applications JEE sont des systèmes multi-niveaux, ils utilisent une approche multi-niveaux pour classifier les patrons JEE. Le catalogue de ces patrons est réparti sur les trois niveaux suivants : le niveau présentation, le niveau métier et le niveau intégration. L'ensemble des patrons JEE est présenté dans le Tableau 1.2.

Tableau 1.2 Catalogue des patrons JEE

Couche Présentation	Couche métier	Couche intégration
- *Intercepting Filter* - *Front Controller* - *Context Object* - *Application Controller* - *View Helper* - *Composite View* - *Service to Worker* - *Dispatcher View*	- *Business Delegate* - *Service Locator* - *Session Facade* - *Composite Entity* - *Transfer Object* - *Transfer Object Assembler* - *Value List Handler* - *Business Object* - *Application Service*	- *Data Access Object* - *Service Activator* - *Domaine Store* - *Web Service Broker*

Alur *et al.* (2003) décrivent chaque patron JEE de façon uniforme à travers six rubriques:

(1) Le problème : présente les problèmes résolus en appliquant ce patron.

(2) Les forces : décrivent les raisons et les justifications derrière le choix de l'application de ce patron.

(3) La solution : présente la structure de la solution du patron sous forme de deux diagrammes UML (*Unified Modeling Language*) ainsi que les différentes stratégies utilisées pour implémenter ce patron.

(4) Les conséquences : listent les avantages et les inconvénients résultants de l'application du patron,

(5) Un exemple de code : présente un exemple d'implémentation du patron. Cette rubrique peut être incluse parfois dans la sous-partie stratégie de la rubrique solution.

(6) Les patrons apparentés : listent les patrons JEE ou les patrons '*Gang of Four'* (GoF) qui sont en relation avec le patron JEE décrit.

Une brève description du rôle de chaque patron est fournie en ANNEXE I. Des explications supplémentaires sur certains patrons JEE sont présentées dans les chapitres suivants.

Exemple de patron JEE : « *Data Acccess Object (DAO)* »

Le patron « *Data Access Object (DAO)* » a pour but d'abstraire et d'encapsuler la partie relative à l'accès aux sources de données. Il gère la connexion aux sources de données afin

d'obtenir ou de stocker les données en masquant les détails d'implémentation des fonctionnalités liées à l'accès à ces sources de données. Ceci empêche le changement des composants métiers ou des clients en cas de changement de l'implémentation des fonctionnalités relatives à l'accès aux données. La structure du patron « *DAO* » est montrée dans la Figure 1.5.

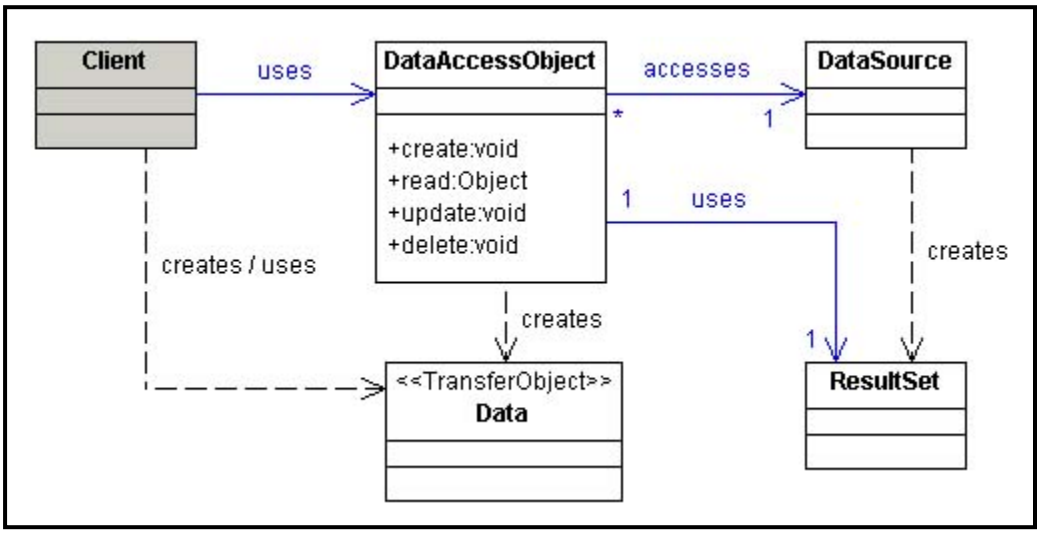

Figure 1.5 Diagramme de classes du patron « *Data Access Object* (*DAO*) »
Tirée de Alur et al (2003, p.463)

Le 'client' est un composant qui a besoin d'accéder à une source de données pour obtenir ou sauvegarder des données. Il est généralement un composant de la couche métier, par exemple un « *Business Object* » ou un « *Session Facade* »[5]. Le client va accéder au composant '*DataAccessObject'* qui joue le rôle principal du patron « *DAO* ». Ce composant regroupe l'implémentation des fonctionnalités qui permettent l'accès à la source de données. Il contient l'implémentation des opérations d'insertion, de recherche, de mise à jour et de suppression des données. Les données manipulées sont stockées dans le composant '*Data Source*' qui représente la source de données, qui peut être par exemple une base de données ou un document *Extensible Markup Language* (XML). Le résultat de l'exécution d'une requête est sauvegardé dans le composant '*ResultatSet*'. Les données échangées entre les

[5] Ces patrons sont décrits brièvement dans l'ANNEXE I.

clients et la source de données (c-à.-d. les données envoyées par les clients ou reçues de la part du '*DataAccessObject')* décrivent le contenu du composant '*Data*' qui peut être un « *Transfer Object* ». Ce dernier est un patron JEE qui vise à minimiser le transfert de données à travers différentes couches. Dans le cas où l'application utilise un seul type de source de données, le patron « *Factory Method* » peut être appliqué pour implémenter le patron « *DAO* ». Pour le cas où l'application utilise plusieurs types de sources de données, le patron « *Abstract Factory* » peut être appliqué pour implémenter ce patron.

L'utilisation du patron « *DAO* » a plusieurs conséquences qui incluent : (i) la transparence d'accès à la source de données (car les détails de l'implémentation sont masqués dans le « *DAO* ») ; (ii) la facilité de changer la source de données, par exemple en facilitant la migration vers une nouvelle implémentation de base de données ; et (iii) la réduction de la complexité du code client. Parmi les aspects négatifs du patron « *DAO* » présentés par Alur *et al.* (2003), nous citons le fait que l'utilisation des patrons « *Factory Method* » et « *Abstract Factory* » pour implémenter ce patron augmente la complexité de la conception.

Le patron « *DAO* » se comporte comme un « *Adaptateur* » entre l'application et la source de données. De plus, il joue le rôle d'un *broker* entre les clients et les serveurs des systèmes distribués en découplant les composants de l'application des interactions avec les services externes.

Dans les sections suivantes, nous présentons des travaux qui se sont intéressés à l'utilisation des patrons et à l'évaluation de leur impact sur la qualité des applications. En particulier, nous abordons dans la section 1.6 des travaux qui portent sur les patrons de conception en général. Dans la section 1.7, nous discutons les travaux qui portent spécifiquement sur les patrons JEE.

1.6 Études sur l'utilisation et l'impact des patrons de conception

Plusieurs travaux se sont intéressés à l'utilisation des patrons de conception et à l'évaluation de leur impact sur la qualité des applications (e.g., (Bieman et al., 2003), (Jeanmart et al., 2009)). Ces travaux se distinguent par leurs objectifs et par les méthodes d'analyse utilisées.

1.6.1 Travaux basés sur l'analyse des classes et leurs rôles dans les patrons

Certains travaux (e.g., (Di Penta et al., 2008) et (Khomh, Guéhéneuc et Antoniol, 2009)) ont analysé l'impact des patrons à travers l'évaluation des classes qui jouent des rôles dans des occurrences de ces patrons.

Di Penta *et al.* (2008) présentent une étude empirique qui vise à analyser l'évolution des classes qui jouent des rôles différents dans les patrons de conception. Le but de cette étude est d'identifier s'il existe des rôles de patrons qui sont plus sujets à changement que d'autres. Cette étude porte sur 12 patrons de conception et elle est effectuée sur trois systèmes Java, open source, de différentes tailles et de différents domaines qui sont *JHotDraw*, *Xerces*, et *Eclipse JDT*.

Di Penta *et al.* ont utilisé l'outil '*Design Motif Identification Multi-layered Approach*' (DeMIMA) pour identifier les patrons de conception dans les systèmes analysés. Cette approche n'identifie pas seulement les classes qui décrivent les occurrences de patrons, mais aussi les différents rôles joués par ces classes. Nous résumons les observations retenues de cette étude dans le Tableau 1.3.

Tableau 1.3 Les observations retenues du travail de Di Penta *et al.*

Patron de conception	**Observations**
Prototype	Aucune preuve n'a été trouvée
Singleton	⇨ L'intention de l'utilisation de ce patron est de fournir un point d'accès unique plutôt que de rendre le système robuste aux changements. Donc, aucun impact de l'utilisation de ce patron sur les changements
Fabrication Abstraite	Les entités concrètes ('*ConcreteFactory*' et '*Product*') sont plus fréquemment changées que les entités abstraites ('*AbstractFactory*' et '*AbstractProduct*'). ⇨ L'intuition derrière l'utilisation de ce patron est confirmée
Adaptateur	L'entité '*Adaptateur*' présente plus de changements que les entités '*Target*' ⇨ L'intuition derrière l'utilisation de ce patron est confirmée
Commande	Les '*Receiver*' sont les entités les plus sujettes à changement que les entités '*Command*' ⇨ L'intuition derrière l'utilisation de ce patron est confirmée
Composite	Les composants '*Leaf*' sont plus sujets à changement que l'entité '*Component*' et les entités '*Composite*' ⇨ L'intuition derrière l'utilisation de ce patron n'est pas confirmée (aucune différence significative n'est identifiée pour les différents rôles)
Décorateur	Les entités concrètes ('*ComposantConcret*' et '*DecorateurConcret*') sont plus sujettes à changement que les entités '*Composant*' et '*Decorateur*'. ⇨ L'intuition derrière l'utilisation de ce patron est confirmée
Méthode Fabrique	Les entités '*ConcreteProduct*' et '*ConcreteCreator*' sont plus sujets à changement que les entités '*Product*' et '*Creator*' ⇨ L'intuition derrière l'utilisation de ce patron est confirmée
Observateur	Les entités '*Observe*' et '*Observateur*' vont être changées rarement au contraire des entités concrètes ('*ConcreteObserve*' et '*ConcreteObservateur*') qui vont changer en fonction des nouveaux besoins qui apparaissent. ⇨ L'intuition derrière l'utilisation de ce patron est confirmée
État/ Stratégie	Les entités '*Etat'* et '*Strategy*' doivent être les moins sujettes à changement que les entités concrètes ('*ConcreteEtat*' et '*ConcreteStrategy*'). ⇨ L'intuition derrière l'utilisation de ce patron est confirmée
Template Méthode	Les entités '*ConcreteClass*' sont plus sujettes à changement que les entités '*AbstractClass*'. ⇨ L'intuition derrière l'utilisation de ce patron est confirmée
Visiteur	Les composants '*Elements*' sont moins sujettes à changement que les entités '*ConcreteVisitor*'. ⇨ Rien ne peut-être conclure à partir des résultats qui ont été trouvés

Les résultats trouvés confirment souvent le comportement attendu et l'intuition du patron. Il y a, cependant, des cas où les changements observés divergent de cette intuition, par exemple le patron « *Composite* ».

Khomh, Guéhéneuc et Antoniol (2009) présentent une étude empirique dont le but est d'évaluer l'impact des classes jouant un ou deux rôles dans des occurrences de patrons de conception sur la qualité des systèmes logiciels. Cette étude a portée sur six applications java, logiciels libres et de différents domaines, tailles et complexités. Six patrons de conception ont été visés par l'étude : « *Commande* », « *Composite* », « *Décorateur* », « *Observateur* », « *Singleton* », et « *État* ». Comme dans (Di Penta et al., 2008), les occurrences des patrons et les rôles des classes sont identifiées en utilisant l'approche DeMIMA.

Les résultats de cette étude démontrent que les classes jouant plus d'un rôle dans des occurrences de patron ont plus d'impact sur la qualité des applications que celles qui jouent zéro ou un rôle. De plus, ces classes tendent à être moins cohésives, plus couplées, plus complexes et plus fréquemment modifiées.

1.6.2 Travaux portant sur l'évaluation de l'impact des patrons sur la qualité

Bieman *et al.* (2003) présentent une étude empirique qui vise à identifier les effets de l'application des patrons de conception sur les changements qui se produisent lors de l'évolution des systèmes. Cette étude a porté sur plusieurs versions de cinq systèmes dont trois sont des systèmes industriels et deux sont des logiciels libres. Les patrons de conception visés par l'étude sont ceux présentés dans (Gamma et al., 1994) et (Grand, 2002).

La détection et l'analyse d'une occurrence d'un patron se font en trois étapes : 1) chercher dans la documentation (ex. les commentaires dans le code source) des classes associés à un nom de patron ; 2) pour une classe identifiée à la première étape, analyser le modèle de classes du système pour identifier l'ensemble des classes en relation avec cette classes ; et 3)

analyser le modèle de classes composé de cette classe et les classes reliées pour vérifier le degré de conformité de ce modèle à la solution proposée par le patron de conception.

Les résultats de cette étude prouvent qu'il y a une forte corrélation entre la taille d'une classe et la fréquence des changements. De plus, les classes qui font partie d'une occurrence d'un patron ou qui ont été raffinées par héritage, ont tendance à être les plus fréquemment changées dans le système. Cependant, ce dernier constat ne peut être généralisé à tous les systèmes.

Jeanmart et al. (2009) présentent une étude qui vise à évaluer l'impact de l'utilisation du patron « *Visiteur* » sur la compréhension et la modification des programmes. 24 étudiants sont impliqués dans cette étude. Plusieurs des participants ont une expérience en milieu industriel et un niveau de connaissance élevé d'UML et des patrons de conception. Ils ont été divisés en trois groupes. L'étude se base sur un questionnaire audiovisuel qui porte sur trois logiciels libres (*JHotDraw*, *JRefactory* et *PADL*) dans lesquelles le patron « *Visiteur* » est appliqué. En particulier, elle évalue le temps dépensé et les zones d'attention utilisées par chaque participant de chaque groupe pour répondre au questionnaire. Cette évaluation est effectuée par l'entremise d'un oculométre qui enregistre les mouvements afin de calculer la direction du regard et d'identifier les zones d'attentions utilisées pour le calcul et la collecte des données.

L'étude a démontré que les participants qui ont une bonne connaissance d'UML et des patrons de conception ont besoin de moins de temps et d'effort pour comprendre le programme et effectuer les changements que les autres. De plus, une utilisation standard du patron « *Visiteur* » renforce cette tendance. Même si l'étude se limite au patron « *Visiteur* », Jeanmart *et al.* ont conclu que l'utilisation des patrons peut aider, en général, les développeurs à mieux comprendre le programme et à faciliter ses modifications.

Khomh et Guéhéneuc (2008) présentent une étude empirique qui vise à évaluer l'impact de l'utilisation des patrons de conception sur la qualité des systèmes. Un questionnaire a été

fourni à un ensemble de participants qui devaient donner selon leurs expériences, un indice de l'impact de l'application de certains patrons de conception sur la qualité des logiciels. Le questionnaire porte sur trois familles d'attributs de qualité : (i) ceux liés à la conception, ceux liés à l'implémentation et ceux liés à l'exécution. Chaque participant doit accorder une valeur à chaque couple (patron, attribut de qualité). Cette valeur quantifie l'impact du patron sur l'attribut de qualité. Elle peut être : A- très positif, B- positif, C- non significatif, D- négatif, E- très négatif ou F- non applicable (dans le cas ou le participant ne connait pas ou il est incertain de l'impact d'un patron sur un attribut de qualité). Vu que les résultats collectés dépendent du degré de rigueur des participants, les auteurs ont décidé de regrouper les réponses en trois valeurs (A, B= positive, C= neutre et D, E= négative) et de ne pas prendre en compte des réponses de type F. Ils ont sélectionné 20 réponses à étudier. Leur choix est basé sur les participants qui ont de l'expérience avec l'utilisation des patrons lors du développement et de la maintenance des logiciels. Khomh et Guéhéneuc concentrent leur étude sur (i) l'impact de trois patrons (« *Fabrication abstraite* », « *Composite* » et « Poids-*mouche* ») sur l'ensemble des attributs de qualité considérés et (ii) l'impact de tous les patrons de conception sur trois attributs de qualité (la réutilisabilité, la compréhensibilité et l'extensibilité). Les deux tableaux 1.4 et 1.5 résument les résultats de cette étude en symbolisant l'impact positif par (+) et l'impact négatif ou neutre par (-).

Tableau 1.4 Impact de trois patrons de conception
sur les attributs de qualité
Adapté de Khomh et Guéhéneuc (2008, p.276)

Attributs de qualité	**Patrons de conception**		
	Composite	**Fabrication abstraite**	**Poids-mouche**
Extensibilité[6]	+	+	-
Simplicité[7]	+	+	-
Réutilisabilité[8]	+	+	-
Facilité d'apprentissage[9]	+	-	-

[6] Exprime la capacité d'étendre un système.
[7] Exprime la facilité de comprendre un système.
[8] Exprime la réutilisation d'un module dans une autre conception.
[9] Exprime la facilité d'apprentissage d'un système

Attributs de qualité	Patrons de conception		
	Composite	**Fabrication abstraite**	**Poids-mouche**
Compréhensibilité[10]	+	-	-
Modularité[11]	+	+	-
Généralité[12]	+	+	-
Modularité à l'exécution[13]	+	-	-
Évolutivité[14]	-	-	+
Robustesse[15]	-	-	-

Tableau 1.5 Impact des patrons de conception sur trois attributs de qualité
Adapté de Khomh et Guéhéneuc (2008, p.277)

Patrons de conception	Attributs de qualité		
	Extensibilité	**Compréhensibilité**	**Réutilisabilité**
Fabrication abstraite	+	-	+
Monteur	+	+	-
Méthode fabrique	+	-	+
Prototype	+	+	+
Singleton	-	+	-
Adaptateur	+	-	+
Pont	+	+	-
Composite	+	+	+
Décorateur	+	-	-
Façade	+	+	-
Poids-mouche	-	-	-
Proxy	-	-	+
Chaine de responsabilité	+	-	+
Commande	+	-	-
Interpréteur	+	+	+

[10] Exprime la facilité de comprendre un système.
[11] Exprime l'indépendance de l'implémentation d'un système.
[12] Exprime la capacité du système de fournir une large gamme de fonctions à l'exécution.
[13] Exprime l'indépendance des fonctions du système les uns des autres à l'exécution.
[14] Exprime la capacité du système de s'adapter aux changements.
[15] Exprime la capacité de système à continuer de fonctionner correctement dans des conditions anormales.

Patrons de conception	Attributs de qualité		
	Extensibilité	**Compréhensibilité**	**Réutilisabilité**
Itérateur	+	+	+
Médiateur	+	+	-
Mémento	-	-	-
Observateur	+	-	+
État	+	+	-
Stratégie	+	+	-
Template Méthode	+	-	+
Visiteur	+	-	-

En nous référant au Tableau 1.4, nous constatons que le patron « *Composite* » impacte positivement la majorité des attributs de qualité. La facilité d'ajouter de nouveaux objets et la simplicité du client expliquent bien l'impact positif de ce patron sur l'extensibilité et la simplicité. Alors que la difficulté de restructurer un objet composite explique l'évaluation négative de son impact sur l'évolutivité et la robustesse. Pour le patron « *Fabrication Abstraite* », le nombre d'attributs de qualité impactés positivement est le même que celui des attributs impactés négativement. La facilité d'ajouter de nouveaux descendants sans changer les classes qui les utilisent et la séparation entre la création et l'utilisation des objets expliquent l'impact positif de l'utilisation de ce patron sur l'extensibilité, la simplicité et la généralité. De plus, le changement des classes concrètes sans modifier le code des classes qui les utilisent affecte positivement la modularité et la réutilisation. L'impact négatif de l'utilisation de ce patron sur la modularité et l'évolutivité des systèmes peut être expliqué par la difficulté d'écrire une implémentation parfaite de ce patron. La complexité introduite en utilisant le patron «*Fabrication abstraite*» affecte négativement la facilité d'apprentissage et la compréhension.

Malgré que le patron « *Poids-mouche* » permet à plusieurs objets de travailler simultanément (ce qui affecte positivement l'évolutivité), il est compliqué et dédié à un problème particulier. Cela explique l'impact négatif de ce patron sur la facilité d'apprentissage, la compréhensibilité et la réutilisation.

Les résultats reportés au Tableau 1.5 qui regroupe l'impact de tous les patrons de conception sur trois attributs de qualité, indique que l'utilisation de ces patrons impacte négativement plusieurs attributs de qualité. Malgré que les patrons de conception résolvent plusieurs problèmes, ils n'améliorent pas toujours la qualité des systèmes. De façon générale, les auteurs ont constaté que : (i) l'extensibilité est améliorée en appliquant les patrons de conception et (ii) la réutilisabilité et la compréhensibilité sont impactées négativement par l'utilisation des patrons. Par conséquent, ils recommandent d'appliquer prudemment les patrons.

1.6.3 Études systématiques analysant l'état de l'art sur l'utilisation et l'évaluation des patrons

Des études récentes ont porté sur l'analyse et le survol de l'état de l'art sur l'évaluation de l'impact des patrons de conception sur la qualité des systèmes.

Zhang et Budgen (2012) ont proposé une étude systématique de la littérature dont les buts sont (i) d'identifier les patrons GoF qui ont été évalués de manière empirique en pointant vers les conclusions faites sur l'impact de l'utilisation de ces patrons sur la maintenance des systèmes et (ii) d'identifier les études nécessaires pour pallier aux lacunes identifiées dans la littérature.

Cette étude est présentée sous forme d'une cartographie dont le but est d'identifier les études pertinentes sur l'utilisation des patrons GoF et qui ont été réalisées entre 1995 et 2009. La sélection des études pertinentes s'est faite en combinant une recherche électronique couvrant la période ciblée en utilisant un ensemble de mots clés et six moteurs de recherche et une recherche manuelle dans certaines revues.

611 documents sont retenus après cette recherche dont 219 articles qui reportent des études empiriques. Ces articles sont filtrés en se basant sur : (1) le titre du document, (2) la lecture du résumé et (3) la lecture de tout le document. Vu le nombre limité de documents retenus après cette sélection (uniquement 10 papiers), Zhang et Budgen ont élargi l'ensemble des

études à analyser en ajoutant sept rapports d'expériences qui décrivent un ensemble d'observations sur les patrons. Tous les documents recensés discutent les patrons GoF.

Cette étude souligne le nombre élévé de travaux visant à identifier et documenter les patrons ou détecter leur présence dans les logiciels. Très peu de travaux se sont penchés sur l'évaluation de l'impact des patrons sur la qualité des systèmes. Aussi, cette étude souligne que très peu de patrons ont été analysés dans ces travaux. En effet, les patrons « *Composite* », « *Observateur* » et « *Visiteur* » ont été les plus analysés dans la littérature. Même pour ces patrons, les résultats de ces analyses se limitent à des observations telles que :

- l'utilisation du patron « *Composite* » avec le patron « *Visiteur* » est bénéfique et elle conduit à moins d'erreurs;
- la récursivité introduite par le patron « *Composite* » entraine des problèmes pour certains développeurs;
- le patron « *Visiteur* » facilite la modification, mais il complique le code.

Zhang et Budgen suggèrent d'approfondir les études visant l'analyse de l'utilisation des patrons et des conséquences de leurs utilisations. Ils estiment que l'évaluation de l'impact de l'utilisation des patrons sur la qualité est un des sujets les plus importants à aborder dans de travaux futurs.

Ali et Elish (2013) présentent une étude qui vise à identifier les travaux qui ont étudié les liens entre les attributs de qualité et les patrons de conception et à synthétiser l'impact de l'utilisation des patrons sur la qualité des logiciels en se référant à ces travaux.

Cette étude analyse 17 travaux. Les attributs de qualité abordés dans ces études incluent : (i) la maintenabilité, (ii) l'évolution et la prédisposition aux changements (« *Evolution and Change-proneness* »), (iii) la performance et (iv) la prédisposition aux fautes («*Fault-proneness* »). Selon (Ali et Elish, 2013), la majorité de ces travaux ont porté sur la maintenabilité des logiciels en évaluant l'impact de l'utilisation des patrons de conception sur la qualité de l'application soit au niveau système ou au niveau classe.

Ali et Elish ont constaté qu'il n'y a pas de consensus parmi les travaux analysés sur l'impact des patrons de conception sur les attributs de qualité des logiciels : certains travaux ont rapporté un impact positif ou neutre tandis que d'autres ont signalé un impact négatif. L'incompatibilité des résultats reportés dans ces travaux peut être due à l'expérience des personnes impliquées dans les expérimentations et aux approches utilisées pour évaluer l'impact. Le Tableau 1.6 regroupe tous les résultats reportés dans tous les travaux analysés. Le chiffre entre parenthèses indique le nombre de travaux qui ont rapporté la même observation.

Tableau 1.6 Impact de l'utilisation des patrons de conception sur les attributs de qualité
Adapté de Ali et Elish (2013, p.6)

Attribut de qualité	**Patrons de conception**	**Impact**
Maintenabilité	Ne spécifie pas les patrons	Négatif (3) / Positif (2)/ Neutre (1)
	Observateur	Positif (1)
	Décorateur	Positif (1)
	Visiteur	Négatif (2)
	Composite	Négatif (1) (mais pas autant que le patron Visiteur)
	Fabrication abstraite	Neutre
Évolution et prédisposition aux changements	Ne spécifie pas les patrons	Négatif (3)
Performance	État	Positif (1) (les systèmes complexes) Négatif (1) (les systèmes simples)
	Façade	Fournit une meilleure performance que le patron Commande

Attribut de qualité	Patrons de conception	Impact
Prédisposition de fautes	Ne spécifie pas les patrons	Négatif (1)/ Neutre (1)
	Adaptateur	Négatif (1)
	Observateur	Positif (1)/ Négatif (1)
	Singleton	Négatif (1)
	Méthode fabrique	Positif (1)
	Template	Tendance pas claire
	Décorateur	Non connu

Le Tableau 1.6 montre qu'il y a un consensus sur le fait que l'impact des patrons de conception sur la maintenabilité et sur l'évolution et prédisposition aux changements est négatif alors qu'il n'y en a pas pour le cas de la performance et de la prédisposition aux fautes. Deux raisons empêchent de conclure sur l'impact exact des patrons de conception sur la performance. Ces raisons sont : (i) les patrons de conception ne visent pas l'amélioration de la performance et (ii) il existe peu de travaux qui portent sur cet attribut de qualité. Pour le cas de la prédisposition de fautes, la divergence des résultats des travaux ne permet pas de conclure quel est l'impact des patrons sur cet attribut de qualité. Enfin, cette étude montre que les patrons de conception GoF n'ont pas été tous couverts dans la littérature et même pour les patrons traités, il existe une claire divergence entre les travaux qui abordent l'impact de ces patrons.

1.7 Approches centrées sur les patrons JEE

À l'exception de Mouratidou *et al.* (2010), nous n'avons trouvé aucun travail n'a porté sur l'évaluation de l'impact des patrons JEE sur la qualité des applications. Nous présentons donc les travaux qui ont abordé les patrons JEE de façon générale.

Gilart-Iglesias *et al.* (2005) présentent la façon dont les patrons JEE peuvent être utilisés pour implémenter une architecture Model-View-Controller (MVC). Cette dernière organise une application en trois composants : 1) la vue, qui représente l'interface à travers laquelle l'utilisateur interagit avec l'application ; 2) le modèle, qui représente les données et les traitements d'affaires ; et 3) le contrôleur, qui intercepte les actions faites à travers la vue

pour les traduire en requêtes vers le modèle et modifier la vue pour afficher les résultats de ces requêtes. Cette architecture permet de modifier les vues sans modifier le modèle.

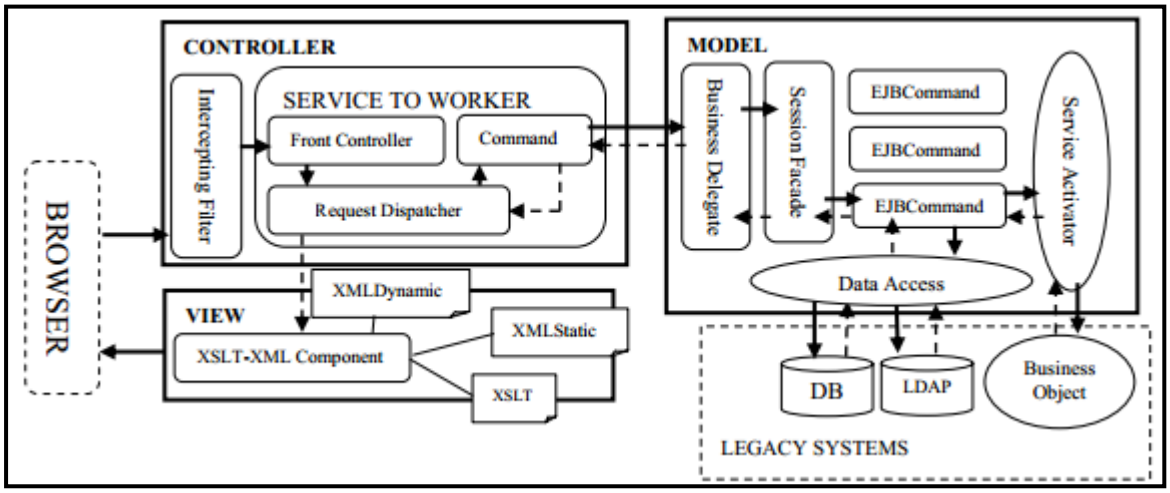

Figure 1.6 Modèle en couches qui utilise les patrons JEE
Tirée de Gilart-Iglesias *et al.* (2005, p.354)

Comme montré par la Figure 1.6, trois patrons JEE et un patron de conception (« *Command* ») sont utilisés pour implémenter le contrôleur. L'« *Intercepting Filter* » gère les demandes en les rejetant ou en les transmettant au « *Front-Controller* ». Ce dernier analyse et vérifie les demandes (par exemple la validation de l'authentification, de la session, de l'autorisation,…..). Il joue le rôle d'un point d'accès unique au système qui assure la transmission des demandes au « *Request Dispatcher* » en lui fournissant la vue à afficher et l'action à effectuer. Le patron « *Request Dispatcher* » a pour objectif d'identifier le service à exécuter. Le patron « *Command* » est utilisé pour encapsuler les informations reliées aux actions à exécuter. Ces informations vont être envoyées vers le modèle où elles vont être exécutées.

Le modèle représente la couche métier et il est implémenté en utilisant cinq patrons JEE. Le « *Business Delegate* » qui cache au contrôleur les détails de l'implémentation des services qui sont décrits dans la « *Session Facade* ». Ce dernier va utiliser des « *EJBCommands* » pour effectuer les traitements. Les « *EJBCommands* » encapsulent les informations dans des

objets *serialisables*. Les deux derniers patrons « *Data Access Object* » et « *Service Activator* » assurent la communication entre le modèle et les systèmes externes tels que les bases de données. Le patron « *DAO* » encapsule les fonctionnalités relatives à l'accès aux bases de données alors que le patron « *Service Activator* » joue le rôle d'un intermédiaire pour accéder à d'autres services commerciaux de façon asynchrone.

La vue contient les interfaces qui vont être exposées à l'utilisateur. Les résultats des requêtes (exécutées par le modèle) sont transmis à la vue en utilisant le patron « *Request Dispatcher* ». Ces résultats sont en format *eXtensible Markup Language* (XML).

Hammouda et Koskimies (2002) proposent, quant à eux, un modèle qui décrit les relations entre 12 patrons JEE. Le modèle proposé est illustré dans la Figure 1.7.

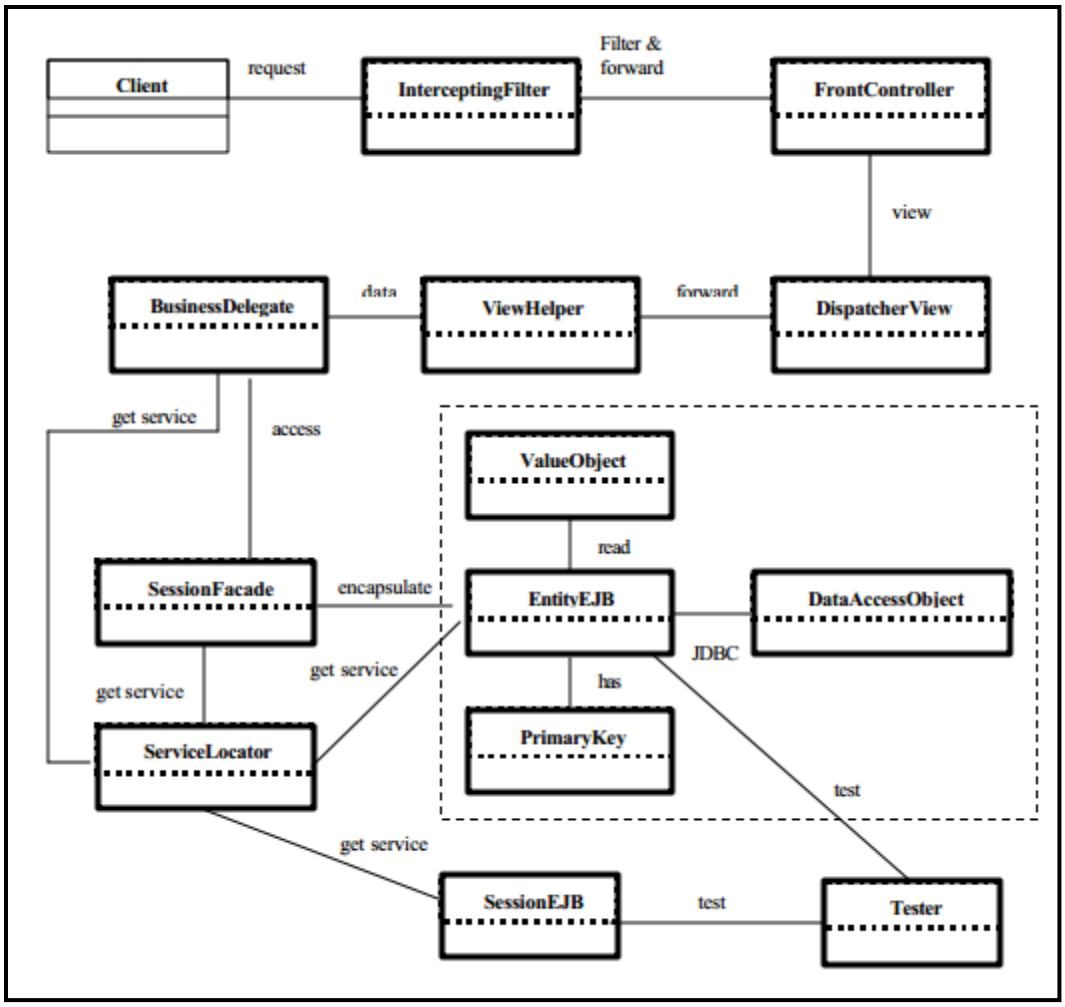

Figure 1.7 Modèle proposé par Hammouda et Koskimies
Tirée de Hammouda et Koskimies (2002, p.252)

Ce modèle est une version modifiée du modèle proposé par Alur *et al.* (2003). En effet, Hammouda et Koskimies ont introduit de nouveaux patrons JEE, par exemple les patrons « *Primary Key Pattern* » et « *Tester Pattern* ». De plus, ils ont renommé le patron « *Value Object* » en « *Transfer Object* ».

Hammouda et Koskimies ont conservé la même classification des patrons JEE proposée par Alur *et al.* La couche présentation comporte quatre patrons JEE : l'« *Intercepting Filter* » qui traite les requêtes avant de les transmettre au « *Front Controller* » qui va assurer leurs contrôles, le « *Dispatcher View* » qui va garantir à la fois le contrôle de flux d'exécution et la

navigation entre les vues et finalement le « *ViewHelper* » qui est responsable de l'affichage du résultat de l'exécution de la demande. Les patrons restants à l'exception des patrons « DAO » et « *Tester* » sont des patrons de la couche métier. Le « *Business Delegate* » cache la complexité de l'implémentation des services qui sont décrits dans les « *Sessions Facade*s ». Les patrons « *Business Delegate* » et « *Sessions Facade*s » vont utiliser le « *Service Locator* » pour localiser les composants métier (les beans sessions et les beans entités). Chaque entité qui est représentée par le patron « *EntityEJB* » possède une clé primaire (« *PrimaryKeyPattern* ») et a au moins une valeur (« *ValueObject* »). Le patron « *PrimaryKeyPattern* » décrit les données avec un identifiant unique et le patron « *ValueObject* » unifie le format des données échangées entre le client et la couche métier. Pour le patron « *DAO* », ce dernier regroupe les fonctionnalités relatives à l'accès à la base de données.

Selon Hammouda et Koskimies, les « *EntityEJB* », la clé primaire (« *PrimaryKey* »), les valeurs des entités (« *ValueObject* ») et le « Data Access Object » décrivent les données persistantes d'une application.

Contrairement à ces travaux qui présentent la façon d'utiliser les patrons JEE, le seul travail à notre connaissance qui évalue l'impact de ces patrons sur la qualité des logiciels est celui présenté par Mouratidou *et al.* (2010). Plus précisément, Mouratidou *et al.* (2010) ont évalué l'impact de trois patrons JEE : « *Front Controller* », « *Transfer Object* » et « *Service to Worker* ». Cette étude est effectuée sur une librairie de e-commerce qui est une application électronique développée en utilisant les composants EJB, *Servlet* et des classes Java. Elle a une architecture à trois niveaux. De plus, six métriques ont été choisies pour évaluer la qualité de cette application :

1. CBO (*Coupling Between Objects*) qui mesure le nombre de classes à laquelle une classe est couplée;
2. CM *(Changing Methods*) qui mesure le nombre de méthodes dans le système qui seront affectées par les changements effectués dans une classe spécifique;

3. CHC *(Changing Classes*) qui mesure le nombre de classes clientes affectées par les changements effectués dans les classes serveurs;
4. RFC *(Response For Class*) qui mesure le nombre de méthodes invoquées pour répondre à une demande;
5. WMPC1 (*Weighted Methods Per Class 1)* qui additionne la complexité cyclomatique de toutes les méthodes d'une classe, à titre indicatif, la complexité cyclomatique représente le nombre de chemins indépendants possibles pour exécuter une méthode;
6. WMPC2 (*Weighted Methods Per Class 2)* qui mesure la complexité d'une classe en se basant sur le nombre de méthodes et de paramètres utilisés dans cette classe (plus le nombre est important plus la classe est complexe).

Pour évaluer l'impact des patrons JEE sur la maintenabilité, Mouratidou *et al.* ont analysé l'évolution des métriques après l'extension de l'application étudiée en appliquant les trois patrons JEE ciblés. Autrement dit, ils ont comparé les mesures de l'application avant et après l'implémentation des patrons ciblés. Ces derniers ont été appliqués pour résoudre des problémes identifiés dans l'application, par exemple : la duplication du code source, la nécessité de vérifier les droits d'accès, l'existence d'une grande quantité de données transmises entre la couche métier et la couche présentation, et la séparation des données qui vont être affichées des actions qui vont être exécutées et des fonctionnalités.

Les observations retenues par cette étude sont :

- l'utilisation du patron « *Front Controller* » élimine le code dupliqué ce qui facilite la maintenance, réduit les potentiels d'erreurs et entraine la diminution du couplage et de la complexité ;
- l'utilisation du patron « *Transfer Object* » facilite l'échange des données entre les couches et minimise le nombre d'accès à la base de données, ce qui entraine une diminution de la complexité ;

- l'utilisation du patron « *Service to Worker* » permet d'identifier l'action appropriée qui va être exécutée et la page JSP qui va être affichée, ce qui entraine la diminution du couplage et de la complexité.

De façon générale, les auteurs estiment que l'utilisation des trois patrons JEE (« *Front Controller* », « *Transfer Object* » et « *Service to Worker* ») influence positivement le couplage et la complexité, ce qui améliore la flexibilité, l'extensibilité et la réutilisabilité. Cette observation implique une maintenance plus facile de l'application et une réduction des coûts et des efforts consacrés au développement et à la maintenance.

1.8 Conclusion

L'évaluation de l'impact des patrons sur la qualité des applications est un sujet très important à traiter dans le domaine de génie logiciel (Zhang et Budgen, 2012). Certains travaux (Di Penta et al., 2008) et (Khomh, Guéhéneuc et Antoniol, 2009) ont analysé l'impact des classes qui jouent différents rôles sur la qualité des applications alors que d'autres travaux se sont intéressés à évaluer l'impact de l'utilisation soit des patrons de conception (Jeanmart et al., 2009) et (Khomh et Guéhéneuc, 2008) ou des patrons JEE (Mouratidou et al., 2010) sur la qualité des applications analysées. Pour la majorité de ces travaux, l'évaluation de l'impact des patrons nécessite de vérifier quels patrons ont été utilisés dans les applications analysées. Pour ce faire, certains travaux utilisent des approches manuelles (Bieman et al., 2003) et (Jeanmart et al., 2009) et d'autres font recours à des outils pour faciliter cette tâche de détection (Di Penta et al., 2008) et (Khomh, Guéhéneuc et Antoniol, 2009). Pour évaluer l'impact des patrons utilisés, certains travaux utilisent un ensemble de métriques (Mouratidou et al., 2010) alors que d'autres se basent sur l'expérience des participants (Khomh et Guéhéneuc, 2008) et (Jeanmart et al., 2009).

De façon générale, il y a peu de travaux qui ont porté sur l'analyse de l'utilisation des patrons (i.e., quels patrons sont vraiment utilisés par les développeurs ?) et sur l'évaluation de leur impact sur la qualité des logiciels. Les quelques travaux qui portent sur l'analyse d'impact

des patrons de conception divergent dans leurs résultats et se concentrent sur un nombre très réduit des patrons. Quant aux patrons JEE, une seule étude, à notre connaissance, a abordé l'analyse de leur impact sur la qualité d'une application. Cette étude s'est limitée à l'analyse d'une seule application simple et à un nombre réduit (3) de patrons JEE, ce qui ne permet pas de généraliser les résultats de cette étude.

L'ensemble de ces observations a motivé notre projet de recherche qui consiste en une étude empirique investiguant l'utilisation des patrons JEE dans les applications et leur impact sur la qualité de ces applications.

CHAPITRE 2

DÉFINITION DE L'ÉTUDE EMPIRIQUE

Dans ce chapitre, nous décrivons la première étape de l'étude empirique. Nous commençons par présenter les questions de recherche auxquelles l'étude essaye de répondre. Ensuite, nous présentons les choix à faire pour répondre aux questions de recherche.

2.1 Questions de recherche

Notre étude vise à répondre aux deux questions de recherche suivantes:

(Q1) : Jusqu'à quel degré les patrons JEE qui supportent la modifiabilité sont-ils utilisés dans les applications JEE?

(Q2) : Quel est l'impact de l'application de ces patrons JEE sur la modifiabilité des applications ?

Afin de répondre aux questions de recherche, nous avons sélectionné un certain nombre de patrons JEE et nous avons analysé un ensemble d'applications. Certaine d'entre elles possèdent plus d'une version disponible pour identifier les patrons JEE appliqués et évaluer leur impact sur la qualité.

Dans le but de répondre à la première question de recherche (Q1), il faut identifier **les patrons JEE** qui supportent la modificabilité et choisir l'ensemble **d'applications JEE** à étudier pour analyser l'utilisation de ces patrons.

Pour ce qui est de la deuxième question (Q2), il faut évaluer la modifiabilité des applications JEE choisies en calculant des métriques et vérifier s'il y a une corrélation entre l'application des patrons JEE et les valeurs des métriques ainsi obtenues. De plus, il faut évaluer la distribution des patrons appliqués dans des versions de quelques applications afin d'évaluer l'impact de cette évolution sur les valeurs des métriques. De ce fait, un ensemble de **métriques** doit d'abord être désigné pour permettre l'évaluation de la modifiabilité.

Pour résumer, notre étude a nécessité le choix: (1) d'un ensemble d'applications JEE à analyser, (2) d'un ensemble de patrons JEE et (3) d'un ensemble de métriques. Ces choix sont discutés dans les sections suivantes.

2.2 Choix des applications JEE

Nous avons sélectionné les applications à analyser en nous basant sur un certain nombre de critères. Ces applications doivent :

i. être des applications JEE;
ii. être des logiciels libres, pour pouvoir accéder au code source et faire des validations manuelles des résultats;
iii. couvrir différents domaines, pour pouvoir identifier le maximum de patrons appliqués (car certains patrons sont utilisés dans des domaines et non pas dans d'autres);
iv. avoir des tailles différentes, pour pouvoir identifier les patrons appliqués dans les applications complexes.

Pour rechercher ces applications, nous avons utilisé quatre sites web de partage les plus connus pour héberger les logiciels libres. Ces sites web sont : '*Sourceforge.net*', '*Java-Source.net*', '*GitHub'* et '*OpenLogic EXchange* (OLEX)'.

Nous avons retenu toutes les applications pour lesquelles nous avons trouvé le code source. La liste des applications JEE est présentée dans le Tableau 2.1. Ces applications sont triées par ordre croissant du nombre de lignes de code source. Les cinq colonnes du Tableau 2.1 indiquent respectivement : le numéro de l'application dans la liste, le nom de l'application accompagné du numéro de la version analysée, le domaine de l'application, la version de JEE utilisée pour le développement de l'application, le nombre de lignes de code source (*Lines of Code* (LOC)) ainsi que le nombre de fichiers sources (Java et JSP). Il est à noter que la version de JEE utilisée pour le développement d'une application est soit citée dans la documentation de l'application, soit déduite de la date de développement ou des versions des

différents composants (*Servlet*, JSP ou EJB). De plus, nous identifions une version JEE donnée par la nomenclature suivante 1.x≈ 1.2|1.3|1.4 et 5+≈5|6|7.

Les versions des applications choisies pour l'analyse sont les plus récentes à l'exception des deux applications '*Glassfish*' et '*WebGoat*' où les versions analysées sont celles dont le code source est disponible.

Tableau 2.1 Liste des applications JEE et leurs caractéristiques

N°	Nom de l'application	Domaine de l'application	Version JEE	Taille (outil=*LocMetrics*)	
				LOC	Nombre de fichiers (Java & JSP)
1	**Peanut** 0.1	*Framework pour développer une application client/Serveur*	J2EE 1.x	1 824	18
2	**Merlidev.cpedev**	Management	JEE 5+	1 825	13
3	**WebGallery**	Application Web	JEE 5+	3 507	49
4	**Joindesk** 1.2 (version minimale)	*Framework pour développer une application web*	J2EE 1.3	3 626	38
5	**J2EE_Bank_Application**	Application Web	J2EE 1.x	4 473	46
6	**Book_shop**	Éducatif	JEE 6	4 816	48
7	**Personalblog** 1.2.6	Communication	J2EE 1.x	5 354	37
8	**Joindesk** 1.2 (version plus complète)	*Framework pour développer une application web*	J2EE 1.3	10 595	137
9	**Java Pet Store** 1.3.1	Commerciale	J2EE 1.3	14 355	118
10	**Locanda**	Commerciale	JEE 5+	16 889	152
11	**Opendating** 0.1.1	Communication	J2EE 1.x	27 907	246
12	**Vaza-mail** 0.1.1	Communication	J2EE 1.4	32 781	161
13	**Paperdog** 0.9	Commerciale	J2EE 1.x	37 256	299
14	**WebGoat** 5.2	Application Web	J2EE 1.x	37 786	145
15	**ChangeSet** 2.2	Application Web	J2EE 1.4	84 144	481
16	**uPortal** 2.1.5	*Framework pour développer des portails web*	J2EE 1.3	97 726	499

N°	Nom de l'application	Domaine de l'application	Version JEE	Taille (outil=*LocMetrics*)	
				LOC	Nombre de fichiers (Java & JSP)
17	**Mvnforum** 1.2.2	Communication	J2EE 1.3	172 917	686
18	**Spring-Framework** 4.0.4	*Framework pour développer des applications Java*	J2EE 1.x	822 964	5413
19	**Glassfish** 2.1	Serveur d'application JEE	JEE 5+	2 219 901	9 817

Le Tableau 2.1 contient 19 applications couvrant sept domaines différents : Framework (5 applications), Communication (4 applications), Commerciale (3 applications), Application web (4 applications), Management (1 application), Éducatif (1 application) et Serveur d'application JEE (1 application).

Voilà une brève description de chacune de ces applications telle que présentée par leurs auteurs:

- ***Peanut***[16] : est un *Framework* qui utilise le protocole *Hypertext Transfer Protocol* (HTTP). Cette application est utile pour développer des applications dont l'architecture peut être de type client/serveur ou de type « *representational state transfer* (REST) ».
- ***Merlidev.cpedev***[17] : est une application JEE de gestion de projets.
- ***WebGallery***[18] : est une application web qui permet de créer des galeries de photos en ligne.
- ***Joindesk***[19] (N°4 et 8) : est un *Framework* pour développer les applications Web. Cette application possède deux versions dont la première est minimale et la deuxième version est complète.

[16] Source Forge. 2013. « peanut ». En ligne. <http://sourceforge.net/projects/tunaep/>. Consulté le 24 avril 2015.
[17] Git Hub. 2014. « merildev/cpedev». En ligne. <https://github.com/merildev/cpedev>. Consulté le 24 avril 2015.
[18]Git Hub. 2014. « webgallery». En ligne. <https://github.com/webgallery/webgallery>. Consulté le 24 avril 2015.
[19] Source Forge. 2013. «Joindesk». En ligne. <http://sourceforge.net/projects/joindesk/?source=directory>. Consulté le 24 avril 2015.

- ***J2EE_Bank_Application***[20]: est une application web qui présente un système bancaire en ligne.
- ***Book_shop***[21] : est une application représentant une librairie en ligne.
- ***Personalblog***[22] : est une application représentant un *blog* personnel. Cette application utilise plusieurs technologies JEE telles que les *servlets* et les pages JSP.
- ***Java Pet Store***[23] : est une application e-commerce qui permet d'acheter des animaux en ligne. Elle a été développée par Sun Microsystems pour illustrer la façon dont la plateforme JEE peut être utilisée pour développer des applications Web.
- ***Locanda***[24] : est une application JEE qui permet la gestion des hôtels.
- ***Opendating***[25] : est un site de rencontres en ligne.
- ***Vaza-mail***[26] : est un serveur de messagerie développé en langage Java.
- ***Paperdog***[27] : est une application Client/Serveur qui gère les versions et l'archivage des documents.
- ***WebGoat***[28] : est une application JEE destinée à gérer des cours de sécurité en ligne.
- ***ChangeSet***[29] : est une application conçue pour le suivi des bugs et la gestion des configurations. ***ChangeSet*** est aussi un système de gestion des changements.

---

[20] m0|interactive. 2004. « J2EE ONLINE BANKING WEB APPLICATION». En ligne. <http://www.m0interactive.com/portfolio/java/j2ee_online_banking_web_application/ >. Consulté le 24 avril 2015.

[21] Git Hub. 2015. « J2EE/book_shop». En ligne. <https://github.com/vlad101/J2EE/tree/master/book_shop>. Consulté le 24 avril 2015.

[22] Source Forge. 2013. « PersonalBlog». En ligne. < http://sourceforge.net/projects/personalblog/?source=directory >. Consulté le 24 avril 2015.

[23] Oracle. En ligne. <http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-eedocs-419425.html#7172-petstore-1.3.1_02-demo-oth-JPR>. Consulté le 24 avril 2015.

[24] Git Hub. 2012. «locanda ». En ligne. <https://github.com/labos/locanda>. Consulté le 24 avril 2015.

[25] Source Forge. 2013. « OpenDating». En ligne. <http://sourceforge.net/projects/opendating/?source=directory>. Consulté le 24 avril 2015.

[26] Source Forge. 2013. « Vaza-mail». En ligne. <http://sourceforge.net/projects/vaza/?source=directory>. Consulté le 24 avril 2015.

[27] Source Forge. 2013. « PaperDog». En ligne. <http://sourceforge.net/projects/paperdog/?source=directory

[28] https://code.google.com/p/webgoat/>. Consulté le 24 avril 2015.

[29] Source Forge. 2013. «QualdevChangeset». En ligne. <http://sourceforge.net/projects/qualdevchangese/?source=navbar>. Consulté le 24 avril 2015.

- ***uPortal***[30] : est un *Framework* développé en utilisant les technologies JEE par les institutions de l'organisation *Java in Administration Special Interest Group* (JASIG). Il sert à développer des portails web Java.
- ***Mvnforum***[31] : est un forum développé en utilisant les technologies JEE.
- ***Spring-Framework***[32] : est un *Framework* JEE conçu pour faciliter le développement des applications Java en particulier les applications JEE en utilisant le modèle MVC.
- ***Glassfish***[33] : est un serveur d'applications JEE. Il est développé par Sun Microsystems pour la plate-forme JEE et maintenant il est commandité par Oracle Corporation.

Pour enrichir l'étude, nous décidons d'étudier différentes versions de quelques applications JEE pour évaluer l'évolution de la distribution des patrons appliqués dans ces versions et d'évaluer l'impact de cette évolution sur les valeurs des métriques.

Deux critères de sélection nous ont guidé pour choisir les applications candidatures à étudier. Ces critères sont :

i. La taille de l'application.
ii. L'existence de plusieurs versions disponibles de l'application.

Nous décidons de limiter notre étude de plusieurs versions à trois applications uniquement. Nous optons pour cette étude les trois applications suivantes '*Joindesk'*, '*mvnForum'* et '*Java Pet Store*'. '*Joindesk'* et '*mvnForum'* sont deux exemples d'applications de petite (respectivement de grande) taille qui possèdent plus qu'une version. '*Java Pet Store*' est

[30]Source Forge. 2013. « uPortal». En ligne. < http://sourceforge.net/projects/uportal/?source=directory>. Consulté le 24 avril 2015.
[31] Source Forge. 2013. « mvnForum». En ligne. < http://sourceforge.net/projects/mvnforum/>. Consulté le 24 avril 2015.
[32] Source Forge. 2013. « Spring Framework». En ligne. < http://sourceforge.net/projects/springframework/>. Consulté le 24 avril 2015.
[33] OpenLogic. 2007. « GlassFish». En ligne. < http://olex.openlogic.com/paquetages/glassfish>. Consulté le 24 avril 2015.

choisi, car elle est connue comme étant une application de référence conçue pour présenter la façon avec laquelle JEE est utilisée.

L'application '*Joindesk*' possède quatre versions disponibles en ligne (0.9, 1.0, 1.1 et 1.2). Nous décidons de les analyser toutes. Pour l'application '*mvnForum*', cette dernière possède sept versions disponibles en ligne. Nous décidons d'analyser le même nombre de versions déjà étudiées dans le cas de l'application '*Joindesk*'. Nous constatons que la numérotation des versions de l'application '*mvnForum*' est de la forme suivante 1.x.x . Vu que nous avons étudié la version 1.2.2, nous décidons de sélectionner les versions qui reflètent des évolutions mineures (1.0, 1.1 et 1.2). Pour le cas de l'application '*Java Pet Store*', cette dernière possède plusieurs versions, mais uniquement le code source de deux versions (1.1.2 et 1.3.1) est disponible. Pour cette raison, nous nous limitons notre étude à analyser ces deux versions.

2.3 Choix des patrons JEE

Il existe 21 patrons JEE recensés dans la littérature (Alur et al., 2003). Comme nous nous intéressons aux patrons JEE qui supportent la modifiabilité, nous nous basons sur les tactiques de modifiabilité pour choisir les patrons JEE à considérer dans notre étude. En fait, nous analysons la description de chaque patron JEE pour identifier quelles tactiques de modifiabilité il supporte.

Par exemple, selon Alur *et al.* (2003) le patron « *Intercepting Filter* » permet de créer des composants faiblement couplés et de petite taille (des filtres) responsables d'un traitement spécifique, ce qui implique une réduction de la taille des modules. De plus, ce patron limite l'accès aux filtres en configurant la liste de filtres à exécuter pour chaque traitement et permet d'éviter l'existence du code dupliqué (car les filtres crées sont indépendants et spécifiques à un traitement particulier), ce qui implique la réduction du couplage. Le patron « DAO » assure l'augmentation de la cohésion en encapsulant les fonctionnalités relatives à l'accès à la source de données. Il permet aussi de réduire le couplage entre la couche métier et la couche persistance en agissant comme intermédiaire entre toute l'application et la

source de données. De plus, il permet la réduction de la complexité du code source des composants qui ont besoin d'accéder à la base de données (Refactorisation).

Les Tableaux 2.2, 2.3 et 2.4 résument les résultats de notre analyse en affichant les correspondances trouvées entre les patrons JEE et les tactiques de modifiabilité. Par ailleurs, ces correspondances sont présentées avec plus de détails dans l'ANNEXE I.

Tableau 2.2 Correspondances entre les patrons JEE de la couche présentation et les tactiques de modifiabilité

			Patrons JEE de la couche présentation							
			Intercepting Filter	***Front Controller***	***Context Object***	***Application Controller***	***View Helper***	***Composite View***	***Service to worker***	***Dispatcher View***
Tactiques de modifiabilité	Réduire la taille d'un module		✓			✓	✓		✓	
	Augmenter la cohésion					✓	✓		✓	✓
	Réduire le couplage	Encapsuler			✓	✓	✓		✓	
		Limiter les dépendances	✓							
		Utiliser un intermédiaire					✓	✓	✓	
		Restructurer	✓	✓					✓	
		Abstraire les services communs		✓					✓	
	« Defer Binding »									

Tableau 2.3 Correspondances entre les patrons JEE de la couche métier et les tactiques de modifiabilité

			Patrons JEE de la couche métier								
			Business Delegate	***Service Locator***	***Session Facade***	***Application Service***	***Business Object***	***Composite Entity***	***Transfer Object***	***Transfer Object Assembler***	***Value List***
Tactiques de modifiabilité	Réduire la taille d'un module						✓				✓
	Augmenter la cohésion						✓				✓
	Réduire le couplage	Encapsuler		✓	✓	✓					
		Limiter les dépendances			✓			✓		✓	
		Utiliser un intermédiaire	✓		✓						
		Restructurer	✓	✓	✓	✓		✓	✓		
		Abstraire les services communs									
	« Defer Binding »										

Tableau 2.4 Correspondances entre les patrons JEE
de la couche intégration et les tactiques de modifiabilité

			Patrons JEE de la couche intégration			
			Data Access Object	***Service Activator***	***Domain Store***	***Web Service Broker***
Tactiques de modifiabilité	Réduire la taille d'un module				✓	
	Augmenter la cohésion		✓		✓	
	Réduire le couplage	Encapsuler				
		Limiter les dépendances				
		Utiliser un intermédiaire	✓			✓
		Restructurer	✓			
		Abstraire les services communs				
	« Defer Binding »					

L'étude des 21 patrons JEE nous a permis de retenir 20 patrons JEE qui supportent la modifiabilité. Nous avons éliminé le patron « *Service Activator* » car aucune tactique de modifiabilité n'est appliquée pour implémenter ce dernier. Le patron « *Service Activator* » favorise l'activation des services de façon asynchrone. Il garantit un gain de temps dans le cas où le client n'a pas besoin d'attendre la fin de traitement. Donc, il permet d'améliorer la performance de l'application en termes de temps de réponse plutôt que faciliter la modifiabilité de l'application.

Pour pouvoir détecter les patrons JEE retenus, nous avons cherché des outils qui nous supportent dans cette tâche. Cependant, la détection des patrons JEE n'a jamais été traitée dans la littérature, au meilleur de notre connaissance. La notion de détection de patrons a été plutôt liée aux patrons de conception. Par conséquent, plusieurs outils permettent de détecter les patrons de conception, mais aucun outil ne permet de détecter les patrons JEE.

Cependant, plusieurs patrons JEE sont reliés aux patrons de conception GoF. Par exemple, le patron « *Session Facade* » joue le rôle d'une façade qui encapsule la complexité des interactions entre les composants métiers et il fournit une couche d'accès uniforme aux clients. De ce fait, nous avons décidé d'exploiter les outils de détection des patrons de conception existants. Nous avons donc analysé les 20 patrons JEE retenus pour identifier ceux que l'on peut détecter soit parce qu'ils sont des occurrences de patrons GoF ou bien par d'autres caractéristiques techniques. Par exemple, le patron « *Intercepting Filter* » vise à effectuer les prétraitements et les post-traitements des requêtes client. Sa structure est montrée dans la Figure 2.1.

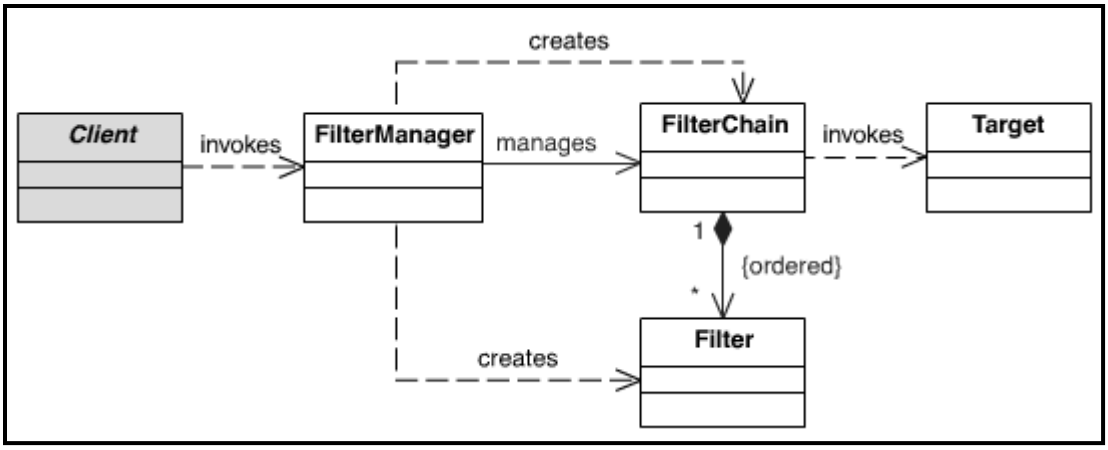

Figure 2.1 Diagramme de classes du patron « *Intercepting Filter* »
Tirée d'Alur et al (2003, p.145)

Une occurrence de ce patron contient un ensemble de classes dont les filtres implémentent l'interface « *javax.servlet.Filter* ». Dans certains cas, il est implémenté en utilisant le patron « *Décorateur* ». Donc pour détecter une occurrence de ce patron, nous supposons que nous pouvons identifier ce patron en détectant toutes les classes qui implémentent l'interface « *javax.servlet.Filter* ».

Le patron « *Session Facade* » vise à regrouper et exposer les composants et les services métier et à contrôler l'accès à ces éléments. Sa structure est montrée dans la Figure 2.2.

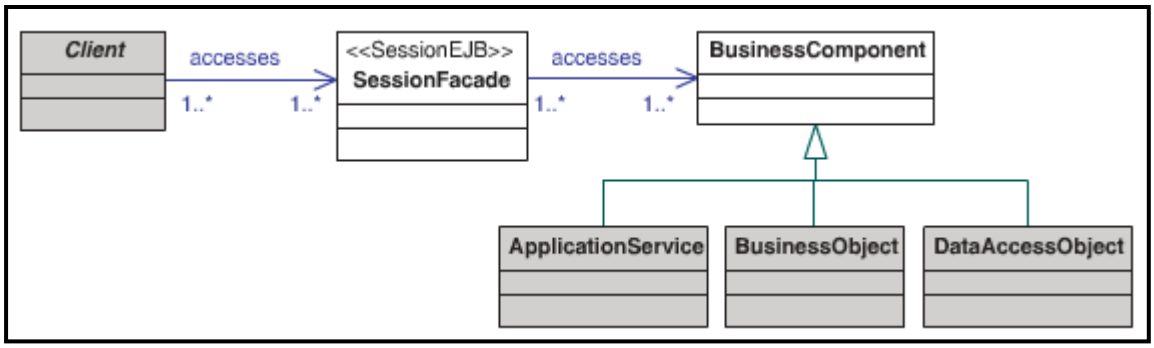

Figure 2.2 Diagramme de classes du patron « *Session Facade* »
Tirée d'Alur et al (2003, p.343)

Ce patron est une occurrence du patron de conception « *Façade* » et en plus la seule classe qui représente ce patron doit implémenter l'interface «*Javax.ejb.SessionBean*». Donc pour détecter une occurrence de ce patron, nous pouvons identifier à la fois les classes qui ont comme rôle 'façade' (un composant du patron façade) et qui implémentent l'interface «*Javax.ejb.SessionBean*». Il est à noter que dans certains cas, la façade peut être implémentée comme étant un « *POJO Facade* ».

Le tableau 2.5 présente la liste des patrons JEE et les caractéristiques qui nous permettent de les détecter. Les colonnes indiquent respectivement le nom de patron JEE et les caractéristiques pour détecter chaque patron.

Tableau 2.5 Liste des patrons JEE dont les caractéristiques nous permettent de les détecter

Patrons JEE	**Caractéristiques**
Intercepting Filter	- Est composé d'un ensemble de composants (**FilterManager**, **FilterChain**, **Filter**) - Type des composants : Classe Java - Est basé sur la notion de filtres dont chaque filtre implémente l'interface « ***javax.servlet.Filter*** » - Peut être implémenté en utilisant le patron « Décorateur » pour gérer la décoration des filtres

Patrons JEE	Caractéristiques
Front Controller	- Un seul composant - Type du composant : *Servlet* / JSP (c'est désirable d'utiliser un *servlet*)
Application Controller	- Un seul composant - Type du composant : Classe Java - Utilise un ***Mapper*** (pour identifier la demande à résoudre et spécifier à quelle vue elle va être déléguée) - Peut-être intégré dans le « *Front Controller* » pour le cas des simples applications.
Business Delegate	- Un seul composant - Type du composant : Classe Java - Contient les références aux « *Session Facade* », « Application Service », « *Data Access Object* ». - Possède une relation 1-à-1 avec le patron « *Session Facade* »
Session Facade	- Un seul composant - Type du composant : Classe Java - Est une session bean - Contient les références aux « *Application Service* », aux « *Business Object* » (entités beans) et aux « DAOs » - Joue le rôle d'une façade
Composite Entity	- Est une composition de beans entités (Beans EJB) - Type du composant : Classe Java - Est un composant EJB
Business Object	- Un seul composant - Type du composant : Classe Java • POJO • Entity bean (Entité EJB) • Implémente l'interface **« *java.io.Serializable* »** - Est une entité simple ou composite - Décrit les beans entités
Application Service	- Un seul composant - Type du composant : Classe Java - Contient les implémentations des traitements - Est utilisé par les « *Session Facade* » ou les « *Business Delegate* ».
Data Access Object	- Un seul composant - Type du composant : Classe Java - Contient les fonctionnalités relatives à l'accès à la base de données (la connexion à la base de données et les requêtes *Structured Query Language* (SQL) pour l'insertion, la modification, la suppression et la sélection). - Joue le rôle d'une façade

D'après le Tableau 2.5, nous constatons que 9 patrons JEE parmi les 20 patrons JEE qui supportent la modifiabilité ont des caractéristiques qui nous permettant de les détecter. En effet, « *Session Facade* » et « *Data Access Object* » sont deux occurrences du patron « Façade ». La majorité des patrons détectables (les 7 patrons restants) ne sont pas des occurrences de patrons de conception ce qui exige l'utilisation d'autres outils et méthodes pour les détecter. Dans le chapitre suivant, nous décrirons en détail le processus de détection ainsi que les méthodes et les outils que nous allons utiliser pour détecter ces patrons.

Nous remarquons que la majorité des patrons détectables sont des patrons de la couche métier, car ils décrivent la logique métier de l'application et ils utilisent des types de composants (EJB, POJO) facilement détectables. Deux raisons font que les patrons ne sont pas détectables. La première est que certains entre eux sont très compliqués, c'est-à-dire composés de plusieurs classes (par exemple le patron *« Domain Store »*) ou de plusieurs patrons JEE (par exemple le patron *« Service to Worker »*). La deuxième raison est que ces patrons peuvent être implémentés en utilisant un type de composant autre que les classes Java, par exemple les pages JSP (par exemple le patron *« View Helper »*).

Pour résumer, nous avons retenu 9 patrons pour la réalisation de notre étude. Ces patrons sont :

- 3 patrons JEE de la couche présentation : « *Intercepting Filter* », « *Front Controller* » et « *Application Controller* »;
- 5 patrons JEE de la couche métier : « *Business Delegate* », « *Session Facade* », « *Composite Entity* », « *Business Object* » et « *Application Service* »;
- 1 patron JEE de la couche intégration : « *Data Access Object* ».

Il est à noter que ces patrons peuvent être utilisés dans la même application, autrement dit ils ne sont pas mutuellement exclusifs.

2.4 Choix des métriques

Comme nous visons à évaluer l'impact des patrons JEE sur la modifiabilité des systèmes, nous nous sommes basés sur les tactiques de modifiabilité implémentées par ces patrons pour choisir les métriques appropriées pour évaluer la modifiabilité des applications analysées.

Nous avons cherché dans la littérature, les métriques les plus utilisées pour évaluer la maintenabilité particulièrement la modifiabilité. Cinq catégories de métriques sont retenues : la taille, le couplage, la cohésion, l'héritage et la complexité. Le Tableau 2.6 illustre les relations entre les catégories de métriques et les tactiques de modifiabilité.

Tableau 2.6 Relations entre les catégories des métriques et les tactiques de modifiabilité

			Catégories des métriques				
			Taille	**Couplage**	**Cohésion**	**Héritage**	**Complexité**
Tactiques de modifiabilité	Réduire la taille d'un module Diviser le module		×				×
	Augmenter la cohésion: Augmenter la cohésion sémantique		×	×	×		×
	Réduire le couplage	Encapsuler		×			
		Limiter les dépendances		×			
		Utiliser un intermédiaire		×			
		Restructurer		×		×	×
		Abstraire les services communs		×		×	
	« Defer Binding »			×		×	

La division d'un module en plusieurs modules de tailles inférieures, qui décrit l'objectif de la première famille de tactiques 'la réduction de la taille d'un module', va affecter la taille ainsi

que la complexité des modules à diviser. Pour la deuxième famille de tactiques, ‘l’augmentation de la cohésion‘, le déplacement des responsabilités pour accroitre la cohésion va affecter la cohésion, la taille, le couplage et la complexité des modules qui ont été modifiés.

L’encapsulation, l’introduction d’une interface ou d’un intermédiaire et la limitation d’accès aux modules vont affecter le couplage des modules changés. L’application des tactiques ‘*refactorisation*’ ou ‘abstraction des services communs’ va affecter l’héritage et le couplage des modules qui subissent des modifications. De plus, la complexité des modules va être affectée en utilisant la tactique de ‘refactorisation’ (restructuration). Quant à la dernière famille de tactiques ‘*Defer binding’* qui peut être basée sur la tactique ‘abstraction des services communs’, son application va affecter l’héritage et le couplage des modules modifiés.

L’étape suivante consiste à identifier un ensemble de métriques pour chaque catégorie. Pour ce faire, nous avons procédé en deux étapes. Dans une première étape, nous avons effectué une recherche sur les travaux réalisés sur ce domaine, par exemple, (Chidamber et Kemerer, 1994), (Briand et al., 2000), (Fontana et al., 2013), (Saraiva, 2013), (Ragab et Ammar, 2010), (Dagpinar et Jahnke, 2003) et (Sheldon, Jerath et Chung, 2002). Ces travaux couvrent les catégories que nous avons identifiées précédemment. Le nombre de métriques correspondant à chaque catégorie dans chacun des travaux recensés dans la littérature est présenté dans le Tableau 2.7. Il y a un total de 134 métriques différentes proposées par ces travaux et qui couvrent les cinq catégories. Dans une deuxième étape, nous avons vérifié la disponibilité d’outils pour calculer ces métriques et nous avons retenu 26 métriques parmi les 134.

Tableau 2.7 Nombre de métriques de chaque catégorie identifié dans les sept articles étudiés

		Les articles étudiés							**Nombre total des métriques différentes**	**Filtrage par disponibilité d'outil**
		(Saraiva, 2013)	(Ragab et Ammar, 2010)	(Dagpinar et Jahnke, 2003)	(Chidamber et Kemerer, 1994)	(Sheldon, Jerath et Chung, 2002)	(Briand et al., 2000)	(Fontana et al., 2013)		
Catégories des métriques	Taille	6	6	3	0	0	6	6	14	5
	Complexité	0	5	0	1	0	0	4	9	3
	Couplage	15	10	33	2	0	28	0	75	6
	Cohésion	15	4	3	1	0	10	2	19	2
	Héritage	5	5	2	2	6	11	1	17	10
									134	26

En analysant les 26 métriques retenues, nous constatons que certaines de ces métriques nous fournissent des informations redondantes, par exemple *Depth of Inheritance Tree* (DIT) et *Average Depth of Inheritance Hierarchy* (ADIH). De ce fait, nous nous limitons à calculer 16 métriques qui couvrent les 5 catégories déjà recensées.

Les 16 métriques sélectionnées sont présentées dans le Tableau 2.8 où la première colonne indique le nom de la catégorie des métriques, la deuxième colonne indique le nom de la métrique, la troisième colonne indique sa définition, la quatrième colonne indique le niveau de calcul des métriques (classe, paquetage ou système) et la dernière colonne contient les outils utilisés pour calculer ces métriques.

Tableau 2.8 Liste des métriques retenues dans la première sélection

Catégorie	**Métrique**	**Définition**	**Calculée au niveau**	**Outil**
Taille	NOM	*Number of Methods*	Système	*CodePro Analytix*[34]
	LOC	*Lines of Code*	Système	*CodePro Analytix*
	NA	*Number of Attributes*	Système	*CodePro Analytix*
	NOC	*Number of Classes*	Système	*CodePro Analytix*
	NOP	*Number of Paquetages*	Système	*CodePro Analytix*
Complexité	WMC	*Weighted Method Per Class*	Système	*CodePro Analytix*
	CYCLO	*McCabe's Cyclomatic Number*	Système	*LocMetric*[35]
Couplage	RFC	*Response For a Class*	Classe	*ckjm*[36]
	CBO	*Coupling Between Objects classes*	Classe	*Ckjm*
	ACAIC	*Ancestor Class-Attribute Import Coupling*	Classe	*Ptidej*[37]

[34] Google Developers. 2001. CodeProAnalytix. Logiciel. En ligne. <https://developers.google.com/java-dev-tools/codepro/doc/>.
[35] locMetrics.com. 2006. LocMetrics. Logiciel. En ligne. <http://www.locmetrics.com/>.
[36] Diomidis Spinellis.2014.Ckjm. Logiciel. En ligne. <http://www.spinellis.gr/sw/ckjm/doc/indexw.html>.
[37] Ptidej Team. 2005. Ptidej. Logiciel. En ligne. <http://www.ptidej.net/tools/>.

Catégorie	Métrique	Définition	Calculée au niveau	Outil
Couplage	ACMIC	*Ancestor Class-Method Import Coupling*	Classe	*Ptidej*
	CA	*Afferent Coupling*	Paquetage	*CodePro Analytix*
	CE	*Efferent Coupling*	Système	*CodePro Analytix*
Cohésion	LCOM1	*Lack of Cohesion in Methods 1*	Classe	*Ptidej*
	LCOM2	*Lack of Cohesion in Methods 2*	Classe	*Ptidej*
Héritage	ADIH	*Average Depth of Inheritance Hierarchy*	Système	*CodePro Analytix*

Nous avons repéré quatre outils permettant de mesurer les métriques représentées dans le Tableau 2.8. Ces outils sont : *CodePro Analytix*[19], *LocMetrics*[20], *Chidamber & Kemerer Java Metrics*[21] (*ckjm*) et *Pattern Trace Identification, Detection and Enhancement in Java*[22] (*Ptidej*). Voici une brève description pour chacun de ces outils :

- ***CodePro Analytix*** est un plug-in Éclipse qui calcule plusieurs métriques de différentes catégories dans les trois niveaux de granularité classe, paquetage et système.
- ***LocMetrics*** est un outil simple qui calcule un nombre limité de métriques au niveau système.
- ***Ckjm*** est un outil qui calcule les métriques orientées objet de CK. Cet outil a été développé en raison du manque des outils complets et fiables permettant de calculer ce genre de métriques.
- ***Ptidej*** est un logiciel libre dédié à l'analyse et à la maintenance des architectures orientées objet. Son but est d'évaluer et d'améliorer la qualité des applications orientées objet (Guéhéneuc et al., 2005). Cet outil calcule les métriques au niveau classe.

Étant donné que les outils peuvent considérer différentes implémentations et interprétations de métriques, nous présentons dans la partie suivante la description de chaque métrique en nous référant à l'outil qui la calcule.

- Nous avons utilisé l'outil *CodePro Analytix* pour calculer neuf métriques, dont cinq métriques de tailles, une métrique de complexité, deux de couplage et une d'héritage:
  - Métriques de taille :
    1. NOM : le nombre de méthodes dans tout le système.
    2. LOC : le nombre total de lignes de code source sans compter les lignes blanches et les commentaires.
    3. NA : le nombre de variables dans tout le système.
    4. NOC : le nombre de classes du système.
    5. NOP : le nombre de paquetages de tout le système.
  - Métrique de complexité :
    1. WMC : la somme des complexités de toutes les méthodes des classes du système. La complexité cyclomatique d'une méthode est le nombre des différents chemins d'exécution dans la méthode (e.g : les chemins créés par des instructions conditionnelles).
  - Métriques de couplage :
    1. CA : le nombre de classes (hors d'un paquetage sélectionné) qui dépendent d'une classe du paquetage sélectionné. (Les points d'entrées à un paquetage)
    2. CE : le nombre de classes d'un paquetage qui dépendent d'une classe d'un autre paquetage. (Les points de sorties de chaque paquetage)
  - Métrique d'héritage :
    1. ADIH : la profondeur moyenne de toutes les classes du système.

- Nous avons utilisé l'outil *LocMetrics* pour calculer une seule métrique de complexité:
    1. CYCLO : La complexité cyclomatique de McCabe de tout le système.

- Nous avons utilisé l'outil *ckjm* pour calculer deux métriques de couplage:
    1. RFC : le nombre de méthodes qui vont être exécutées quand un objet de cette classe reçoit un message.
    2. CBO : le nombre de classes couplées à une classe sélectionnée (en comptant les appels de méthodes, l'accès aux variables, l'héritage et les types et les exceptions retournées).

- Nous avons utilisé l'outil *Ptidej* pour calculer quatre métriques dont deux métriques de couplage et deux de cohésion:
  - Métriques de couplage :
    1. ACAIC : le nombre d'occurrences d'attributs d'une classe dont le type est une classe de l'application.
    2. ACMIC : le nombre de paramètres des méthodes d'une classe dont le type est une classe de l'application.
  - Métriques de cohésion :
    1. LCOM1 : le nombre de paires de méthodes dans une classe qui n'utilisent aucun attribut en commun.
    2. LCOM2 : le nombre de paires de méthodes dans une classe qui n'utilisent aucun attribut en commun - le nombre de paires de méthodes dans une classe qui utilisent au moins un attribut en commun.

Nous observons que lors du calcul des métriques des deux dernières applications de la liste des applications présentée dans le Tableau 2.1, qui sont '*Spring-Framework*' et '*Glassfish*', les outils utilisés n'arrivent pas à calculer certaines métriques (Ceci est relié à un problème de mise à l'échelle). Par conséquent, nous décidons de nous limiter à étudier uniquement les 17 premières applications illustrées dans le Tableau 2.1.

2.5 Conclusion

Nous avons explicité dans ce chapitre la phase préliminaire de l'étude empirique en précisant les questions de recherche et les choix nécessaires permettant d'entamer les phases ultérieures. Les deux questions de recherche auxquelles nous visons à répondre sont : (Q1) Jusqu'à quel degré les patrons JEE qui supportent la modifiabilité sont-ils utilisés dans les applications JEE? et (Q2) : Quel est l'impact de l'application de ces patrons JEE sur la modifiabilité des applications ? Et pour répondre à ces questions nous avons choisis 17 applications JEE à analyser pour détecter neuf patrons JEE et évaluer leurs corrélations avec la modifiabilité en utilisant 16 métriques.

CHAPITRE 3

COLLECTE DES DONNÉES

Dans ce chapitre, nous présentons la deuxième phase de l'étude empirique, à savoir la collecte de données. Nous commençons par présenter le processus de calcul des métriques ainsi que les valeurs des métriques calculées. Ensuite, nous décrivons la démarche suivie et les méthodes utilisées pour la détection des patrons JEE. Nous présentons par la suite les résultats de la détection.

3.1 Calcul des métriques

Dans cette section, nous décrivons en premier lieu le processus suivi pour calculer les métriques. En second lieu, nous présenterons les résultats du calcul des métriques.

3.1.1 Processus de calcul

Pour calculer les valeurs des métriques sélectionnées au chapitre précédent, nous avons suivi le processus décrit par la Figure 3.1.

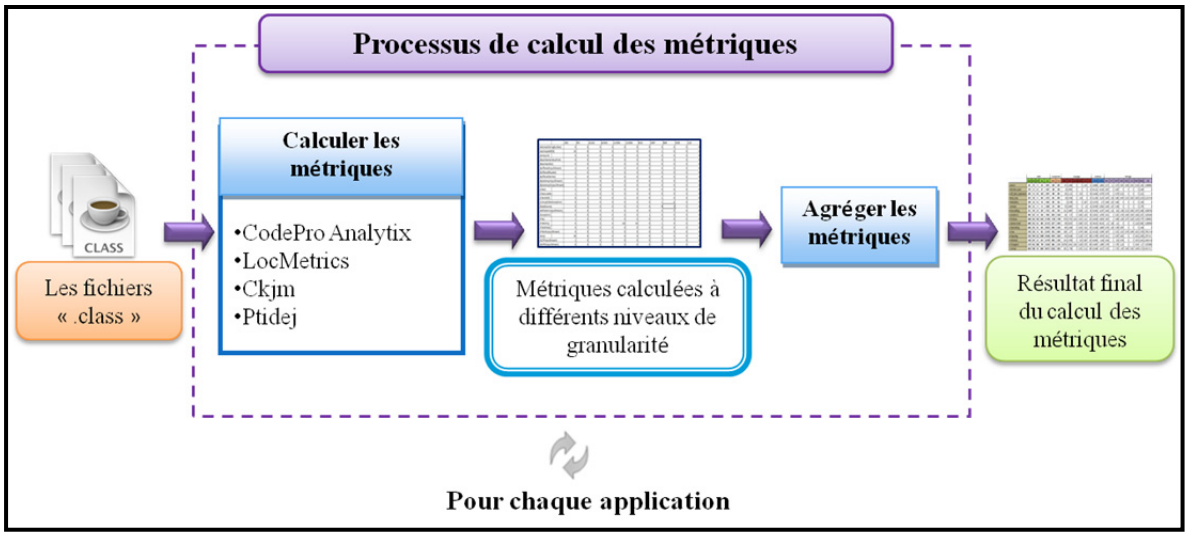

Figure 3.1 Processus de calcul des métriques

La démarche suivie dans ce processus doit être faite pour chaque application. L'entrée du processus de calcul est l'ensemble des fichiers « .class » d'une application à analyser. Il est à noter que pour certaines applications, nous avons dû générer les « .class » à partir du code source en utilisant *Eclipse*. Pour ces applications, cela peut nécessiter une réorganisation manuelle des paquetages à cause de l'existence de plusieurs répertoires «/src » au lieu d'avoir un seul répertoire qui contient tout le code source.

La première étape du processus consiste à calculer les métriques en utilisant les quatre outils présentés dans la section 2.4 qui sont CodePro Analytix, LocMetrics, ckjm et Ptidej. La sortie de la première étape est un fichier Excel qui englobe toutes les métriques calculées. Ces métriques sont calculées à différents niveaux de granularité (système, paquetage et classe). La deuxième étape consiste à normaliser les métriques afin d'assurer la comparabilité et la cohérence des résultats. La normalisation consiste à agréger les mesures calculées au même niveau de granularité système dans le but d'obtenir une seule mesure pour chaque métrique comme résultat de normalisation.

La normalisation des métriques a été faite de plusieurs façons dans la littérature. Par exemple, Fontana *et al.* (2013) ont proposé une normalisation qui consiste à agréger les

mesures calculées au niveau méthode ou au niveau classe par le calcul soit de leur moyenne soit de leur somme. Pour les métriques calculées au niveau système, ils les gardent tels quels. Nous remarquons que les métriques de taille calculées à un niveau autre que le niveau système sont agrégées en calculant la somme, et que toutes les autres métriques sont agrégées en calculant la moyenne.

Roden *et al.* (2007) proposent une autre méthode de normalisation pour les métriques calculées au niveau de granularité classe. La première étape consiste à calculer la moyenne et l'écart type de toutes les valeurs de la même métrique pour les différentes classes. Il s'agit ensuite d'appliquer la formule 3.1 suivante pour chaque valeur :

$$val_n = \frac{val - moyenne}{écart\ type} \tag{3.1}$$

Tirée de Roden *et al.* (2007, p.175)

Avec :

val_n : La valeur normalisée.

val : La valeur initiale de la métrique.

$moyenne$: La moyenne de toutes les valeurs d'une métrique des différentes classes.

$écart\ type$: L'écart type de toutes les valeurs d'une métrique des différentes classes.

Pour normaliser nos mesures, nous nous inspirons du processus de normalisation proposée par Fontana *et al.* (2013), processus assez utilisée dans la littérature. Nous gardons les mêmes valeurs des métriques calculées au niveau système (NOM, NOC, NOP, NA, LOC, WMC, CYCLO, CE et ADIH) et les métriques restantes (CBO, RFC, ACAIC, ACMIC, CA, LCOM1 et LCOM2) vont être agrégées au niveau système en calculant la moyenne.

3.1.2 Résultats de calcul

Le Tableau 3.1 présente les valeurs des 16 métriques pour les 17 applications analysées.

Tableau 3.1 Valeurs des métriques des 17 applications analysées

	Taille					Complexité		Couplage						Cohésion		Héritage
	NOM	NOC	NOP	NA	LOC	WMC	CYCLO	CBO	RFC	ACAIC	ACMIC	CA	CE	LCOM1	LCOM2	ADIH
1-**Peanut** 0.1	86	18	9	64	1057	169	101	2,389	9,2778	0	0	1,333	13	9,48485	2,8485	1,1
2-**Merlidev.cpedev**	178	13	11	87	1187	218	138	3,846	15,692	0	0	2,5	10	96,1212	42,636	2
3-**WebGallery**	206	47	42	118	2107	349	185	2,717	7,7391	0	0,0303	4,18	38	14,4242	4,5758	1
4-**Joindesk** 1.2 (version minimale)	293	37	16	167	2721	631	439	2,158	9,1579	0,013	0,4	1,33	47	41,1236	15,225	2,36
5-**J2EE_Bank_Application**	104	26	11	58	1641	295	156	1,308	10,462	0	0	1,857	23	15,0714	5,75	1,05
6-**Book_shop**	104	16	9	53	1828	272	148	1,188	9,6875	0	0	2,8	14	20,9545	8,5682	1,87
7-**PersonalBlog** 1.2.6	332	47	22	207	3712	676	378	2,394	10,082	0	0,058	1,25	30	30,5652	12,507	2,14
8-**Joindesk** 1.2 (version plus complète)	940	137	33	541	7659	1933	1302	2,744	9,1282	0	0,0833	1,392	96	9,20513	3,0769	1
9-**PetStore** 1.3.1	1696	309	310	1103	17299	3538	1798	3,577	12,381	0,01489	0,0532	3,321	219	28,0638	8,8936	1
10-**Locanda**	1805	152	16	482	11721	2277	1443	3,399	14,101	0	0,0852	7,272	83	194,523	89,142	1,48
11-**Opendating** 0.1.1	1061	246	45	616	10421	1993	1105	3,045	7,4831	0	0,0235	7,414	179	15,0268	5,6409	2,49
12-**Vaza** 0.1.1	1702	160	31	921	21476	3551	2001	2,19	20	0	0,0738	5,666	122	24,5328	9,2049	1
13-**Paperdog** 0.9	1592	299	112	1094	20246	4130	2820	4,325	7,9563	0	0,2276	7,175	200	13,5798	4,9339	1
14-**WebGoat** 5.2	1354	145	22	633	22778	3041	2101	6,32	16,088	0	0,0303	9,687	115	36,6061	14,48	1
15-**ChangeSet** 2.2	1179	406	308	709	33629	3616	2176	4,384	10,971	0,00148	0,0281	6,765	290	4,92467	1,0089	1
16-**uPortal** 2.1.5	4019	499	38	1805	48715	10121	5893	3,893	15,713	0	0,0914	13,18	306	38,5448	14,872	1
17-**mvnForm** 1.2.2	3786	273	26	1884	48411	9291	4554	5,454	15,694	0,0193	0,0856	13,79	222	615,83	298,38	1,92

Nous rappelons que nous avons analysé quatre versions (0.9, 1.0, 1.1 et 1.2) de l'application '*Joindesk*'. Le Tableau 3.2 présente les résultats de calcul des métriques pour les quatre versions.

Tableau 3.2 Valeurs des métriques de quatre versions de l'application '*Joindesk*'

Joindesk (Version minimale)	Taille					Complexité		Couplage						Cohésion		Héritage
	NOM	NOC	NOP	NA	LOC	WMC	CYCLO	CBO	RFC	ACAIC	ACMIC	CA	CE	LCOM1	LCOM2	DIT
Joindesk 0.9	71	14	9	42	592	107	44	1,3333	8,4	0	0	**1,67**	**14**	13,167	3,5833	1,57
Joindesk 1.0	290	36	16	165	2651	616	424	2,666	12,388	0,013	0,413	**1,27**	**46**	46,267	14,493	2,38
Joindesk 1.1	290	36	16	166	2679	624	432	**2,666**	**12,388**	0,013	0,413	1,27	**46**	46,267	14,493	2,38
Joindesk 1.2	293	37	16	167	2721	631	439	**2,621**	**12,189**	0,013	0,4	1,33	**47**	44,56	15,225	2,36

Le Tableau 3.3 présente les résultats de calcul des métriques des quatre versions (1.0, 1.1, 1.2 et 1.2.2) de l'application '*mvnForum*'.

Tableau 3.3 Valeurs des métriques de quatre versions de l'application '*mvnForum*'

mvnForum	Taille					Complexité		Couplage						Cohésion		Héritage
	NOM	NOC	NOP	NA	LOC	WMC	CYCLO	CBO	RFC	ACAIC	ACMIC	CA	CE	LCOM1	LCOM2	DIT
1.0	2917	219	19	1286	43451	7592	3503	**5,95**	**15,89**	0,0103	0,0793	10,69	194	268,21	124,52	1,93
1.1	3773	266	26	1872	48076	9289	4503	**5,47**	**16,44**	0,0196	0,0868	11,3	219	599,95	290,35	1,94
1.2	**3791**	**273**	26	1883	48283	9286	4544	**5,62**	**15,96**	0,0193	0,0856	11,35	222	612,35	296,63	1,92
1.2.2	**3786**	**273**	26	1884	48411	9291	4554	**5,45**	**15,69**	0,0193	0,0856	13,79	222	615,83	298,38	1,92

Nous rappelons que nous avons analysé uniquement deux versions (1.1.2 et 1.3.1) de l'application '*Java Pet Store*'. Les résultats de calcul des métriques des deux versions sont illustrés dans le Tableau 3.4.

Tableau 3.4 Valeurs des métriques de deux versions de l'application '*Java Pet Store*'

Java Pet Store	Taille					Complexité		Couplage						Cohésion		Héritage
	NOM	NOC	NOP	NA	LOC	WMC	CYCLO	CBO	RFC	ACAIC	ACMIC	CA	CE	LCOM1	LCOM2	DIT
1.1.2	864	233	136	410	12219	2246	986	**3,45**	**7,24**	0	0,0092	**5,24**	179	18,581	6,111	2,38
1.3.1	1696	309	310	1103	17299	3538	1798	**3,58**	**12,4**	0,015	0,0532	**3,32**	219	28,064	8,894	1

3.2 Détection des patrons JEE

Dans cette section, nous décrivons en premier lieu le processus de détection que nous avons suivi lors de la détection des patrons JEE. Ensuite, nous présenterons les résultats de la détection.

3.2.1 Processus de la détection des patrons JEE

Nous avons retenu 9 patrons JEE dans l'étape préparation de l'étude empirique. Ces patrons ainsi que leurs relations sont illustrés dans la Figure 3.2.

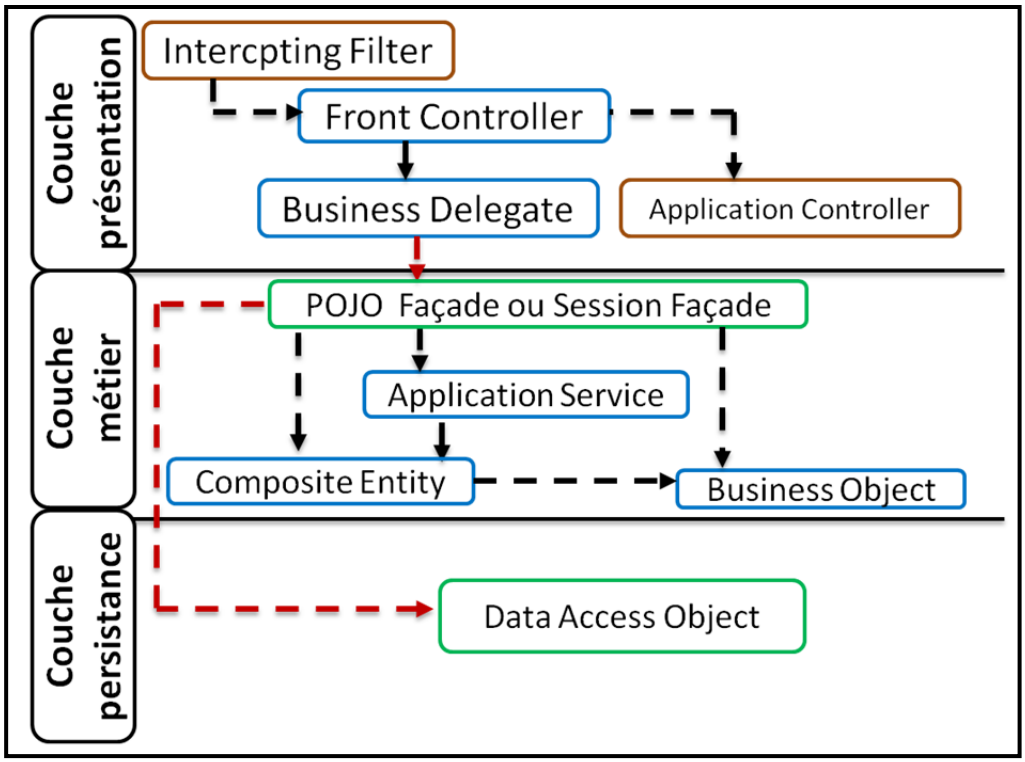

Figure 3.2 Relations entre les patrons JEE retenus pour notre étude

Pour détecter les patrons JEE retenus, nous avons suivi le processus décrit par la Figure 3.3.

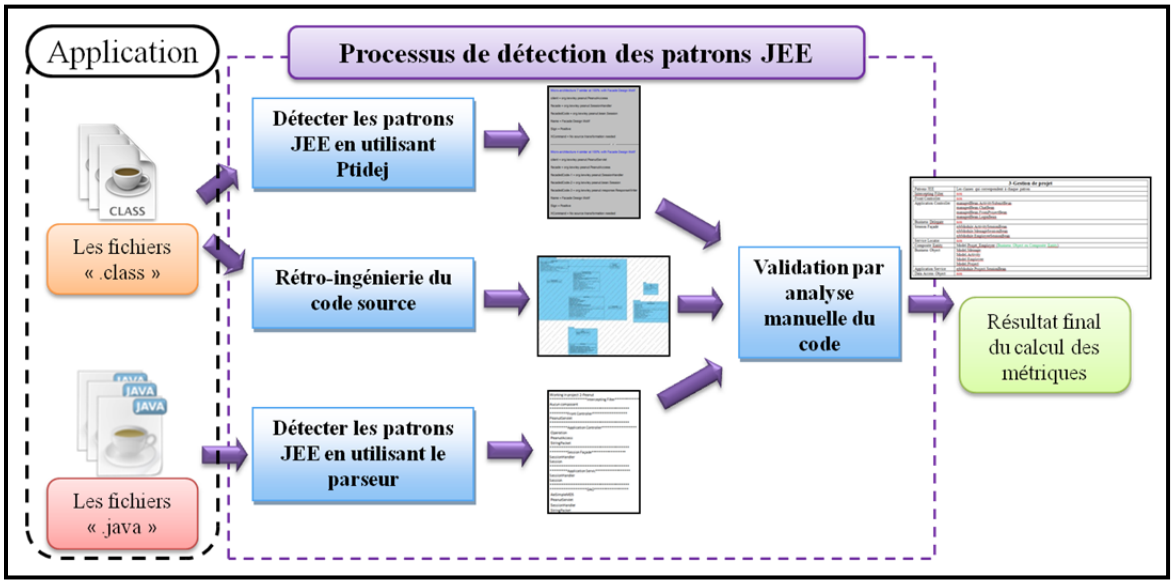

Figure 3.3 Processus de détection des patrons JEE

Les entrées de ce processus de calcul sont les fichiers « .class » et « .java » de chaque application à analyser. La première étape consiste à détecter les patrons JEE en utilisant *Ptidej (Guéhéneuc et al., 2005).* En effet, comme le montre la Figure 3.2, plusieurs patrons JEE sont des occurrences du patron de conception « Façade ». *Ptidej* nous permet de décrire et détecter des occurrences du patron « Façade ». La description de l'occurrence du patron « Façade » est illustrée dans la Figure 3.4 (a). Donc, *Ptidej* va rechercher tous les clients (classes de l'application) qui utilisent une autre classe différente de la première (c'est le composant façade). Cette dernière va utiliser au moins un autre composant (*facadedcode*).

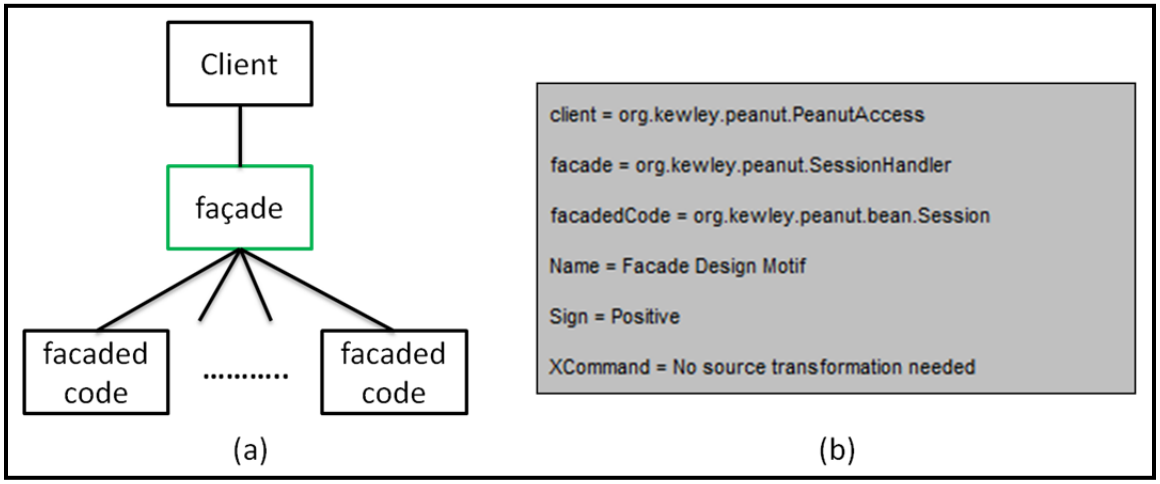

Figure 3.4 Présentation du patron « Façade »
(a) Représentation graphique du patron « Façade », (b) Affichage du résultat de détection fourni par *Ptidej*

Le Tableau 3.5 résume les différentes combinaisons possibles pour détecter des patrons JEE en cherchant des occurrences de « Façade » avec l'outil *Ptidej*. La première colonne du Tableau 3.5 indique le nom du patron JEE et la deuxième colonne présente les patrons JEE qui jouent les rôles des composants du patron « *Façade* ». Par exemple pour le cas du patron « *Front Controller* », ce dernier peut être détecté en identifiant tous les clients dont leurs façades soient un composant qui présente le patron « *Session Facade* », soit un composant qui décrit le patron « *DAO* ».

Tableau 3.5 Différentes combinaisons possibles pour détecter des patrons JEE en cherchant des occurrences de Façade avec l'outil *Ptidej*

Patron JEE	Les composants du patron Façade
Front Controller	• Client : ***Front Controller*** • Façade : *Session Façade* *DAO* • *Facadedcode :* ----

Patron JEE	Les composants du patron Façade
Business Delegate	• Client : ***Business Delegate*** • Façade : *Application Service* *Session Façade* • *Facadedcode :* ----
Session Façade	• Client : ---- • Façade : ***Session Façade*** • *Facadedcode :* *Business Object* *Composite Entity* *DAO* *Application Service*
Data Access Object	• Client : *Session Façade* *Application Service* *Front Controller* • Façade : ***Data Access Object*** • *Facadedcode :* ----
Application Service	• Client : ---- • Façade : *Session Façade* • *Facadedcode :* ***Application Service***
Business Object	• Client : ---- • Façade : *Session Façade* *Application Service* • *Facadedcode :* ***Business Object***
Composite Entity	• Client : ---- • Façade : *Session Façade* *Application Service* • *Facadedcode :* ***Composite Entity***

Pour réduire le nombre de fausses occurrences trouvées (de faux positifs) et d'occurrences positives non trouvées (de faux négatifs), nous avons développé un module d'analyse *'Parseur'* qui exploite des caractéristiques spécifiques aux patrons JEE étudiés. Ce *parseur* parcourt un arbre syntaxique abstrait (*Arbre Syntaxique Abstrait* (AST)) représentant le code source pour y identifier certains nœuds spécifiques. Par exemple des noms contenant des mots particuliers ou des classes implémentant certaines interfaces du *Framework* JEE. Par exemple pour la recherche d'occurrences du patron « *Front Controller* », nous recherchons les classes qui vérifient l'une des contraintes suivantes :

- le nom de la classe contient soit le mot '*Servlet*' ou le mot '*Controller'*;
- une classe qui étend la classe abstraite '*Javax.servlet.HttpServlet*';
- une classe qui contient l'une des deux annotations suivantes '@Controller' ou '@WebServlet'.

Le Tableau 3.6 résume les caractéristiques exploitées par notre *parseur* pour trouver certains patrons JEE étudiés. Ce tableau a été construit de façon incrémentale.

Tableau 3.6 Éléments utilisés pour détecter les patrons JEE en utilisant le parseur développé

Pattrons JEE	**Caractéristiques**				
	Mots clés		**Interface**	**Classe abstraite**	**Annotation**
	Nom de la classe	**Mot dans la classe**			
Intercepting Filter	"*Filter*"		*Javax.Servlet. Filter*		*@WebFilter*
Front Controller	"*Servlet*" "*Controller*"			*Javax.servlet. http.HttpServlet*	*@Controller* *@WebServlet*
Application Controller		"Map"			
Business Delegate					
Session Façade	"Session"		*Javax.ejb. SessionBean*		*@Stateful* *@Stateless* *@Service*
Application Service	"Service"				
Data Access Object	"DAO" "DB"	*"insert", "select", "update", "delete"*			

L'ensemble des occurrences retournées par *Ptidej* et par notre *parseur* sont validées manuellement. Pour ce faire, nous utilisons un outil de rétro-ingénierie d'une part, et une analyse manuelle du code, d'autre part. À titre indicatif, nous avons choisi pour la rétro-ingénierie du code source, l'outil '*Visual Paradigm for Uml*[38]'. L'analyse manuelle du code est un processus laborieux qui consistait à analyser les principales classes du système, les commentaires dans le code et la documentation (Javadoc) de l'application quand elle est disponible.

3.2.2 Résultat de la détection

Le Tableau 3.7 présente le nombre de patrons JEE détectés pour les 17 applications analysées.

[38] Visual Paradigm. Logiciel. En ligne.< http://www.visual-paradigm.com/>.

Tableau 3.7 Nombres de patrons JEE détectés dans chaque application

	Front Controller	Data Access Object (DAO)	Business Delegate	Intercepting Filter	Application Controller	Composite Entity	Business Object	Application Service	Session Façade	POJO Façade	Façade (Session & POJO)
1-**Peanut** 0.1	1	0	0	0	0	0	1	1	0	0	0
2-**Merlidev.cpedev**	0	0	0	0	0	1	4	0	4	0	4
3-**WebGallery**	2	2	0	1	0	0	3	2	0	0	0
4-**Joindesk** 1.2 (version minimale)	0	1	0	0	0	0	5	3	0	0	0
5-**J2EE_Bank_Application**	2	2	0	0	0	0	9	1	0	0	0
6-**Book_shop**	2	3	0	0	0	0	3	3	0	0	0
7-**PersonalBlog** 1.2.6	0	1	0	2	1	0	10	5	0	1	1
8-**Joindesk** 1.2 (version plus complète)	1	0	1	1	0	0	6	4	0	1	1
9-**PetStore** 1.3.1	1	2	2	2	0	2	11	3	8	0	8
10-**Locanda**	0	29	17	2	0	0	23	27	0	0	0
11-**Opendating** 0.1.1	1	12	0	0	0	0	6	19	0	1	0
12-**Vaza** 0.1.1	0	0	0	13	4	0	8	6	0	2	2
13-**Paperdog** 0.9	0	0	0	1	0	9	5	10	6	0	6
14-**WebGoat** 5.2	1	1	0	1	2	1	2	6	1	0	1
15-**ChangeSet** 2.2	0	15	33	0	0	29	18	12	11	0	11
16-**uPortal** 2.1.5	1	26	0	13	0	0	21	13	0	7	7
17-**mvnForm** 1.2.2	2	21	0	2	2	0	24	19	0	2	2

Voilà les Tableaux 3.8, 3.9 et 3.10 qui présentent respectivement le nombre de patrons JEE identifiés pour les différentes versions analysées des applications '*Joindesk'*, '*mvnForum'* et '*Java Pet Store'*.

Tableau 3.8 Nombres de patrons JEE identifiés pour chaque version de l'application '*Joindesk*'

Joindesk (Version minimale)	Front Controller	Data Access Object (DAO)	Business Delegate	Intercepting Filter	Application Controller	Composite Entity	Business Object	Application Service	Façade (Session & POJO)
Joindesk 0.9	0	0	0	0	0	0	4	2	0
Joindesk 1.0	0	1	0	0	0	0	5	3	0
Joindesk 1.1	0	1	0	0	0	0	5	3	0
Joindesk 1.2	0	1	0	0	0	0	5	3	0

Tableau 3.9 Nombres de patrons JEE identifiés pour chaque version de l'application '*mvnForum*'

mvnForum	Front Controller	Data Access Object (DAO)	Business Delegate	Intercepting Filter	Application Controller	Composite Entity	Business Object	Application Service	Façade (Session & POJO)
1.0	2	23	0	2	2	0	23	7	0
1.1	2	20	0	2	2	0	24	13	2
1.2	2	20	0	2	2	0	24	15	2
1.2.2	2	20	0	2	2	0	24	18	2

Tableau 3.10 Nombres de patrons JEE identifiés pour chaque version de l'application '*Java Pet Store*'

Java Pet Store	Front Controller	Data Access Object (DAO)	Business Delegate	Intercepting Filter	Application Controller	Composite Entity	Business Object	Application Service	Façade (Session & POJO)
1.1.2	1	8	2	0	0	0	9	2	3
1.3.1	1	5	2	2	0	2	11	3	8

3.3 Conclusion

Nous avons présenté dans ce chapitre la phase 'Collecte de données ' de l'étude empirique. Nous avons décrit les processus de calcul des métriques et de détection des patrons JEE.

Nous avons présenté les résultats du calcul des métriques et de la détection des patrons JEE de toutes les 17 applications ainsi que les différentes versions des applications '*Joindesk*', '*mvnForum*' et '*Java Pet Store'*. Dans le prochain chapitre, nous discuterons et interpréterons ces résultats.

CHAPITRE 4

ANALYSE DES DONNÉES

Dans ce chapitre, nous discutons les résultats obtenus dans la phase ‘collecte de données’ à la lumière des questions de recherche qui ont été décrites dans le chapitre 2. Nous rappelons que notre première question de recherche visait à évaluer le degré d’utilisation des patrons JEE qui supportent la modifiabilité dans les applications JEE et notre deuxième question de recherche concernait à identifier l’impact de l’utilisation des patrons JEE sur la modifiabilité des applications JEE.

4.1 Jusqu’à quel degré les patrons JEE qui supportent la modifiabilité sont-ils utilisés dans les applications JEE?

Dans cette section, nous présentons une analyse des fréquences d’utilisations des patrons, suivie d’une analyse des cooccurrences des patrons JEE dans les applications analysées. Enfin, nous interprétons la distribution des patrons par domaine d’application.

4.1.1 Fréquences d’utilisation des patrons

La Figure 4.1 montre le pourcentage d’utilisation de chaque patron JEE étudié dans les 17 applications analysées. Afin d’identifier le degré d’utilisation de chaque patron, nous mesurons le pourcentage de l’utilisation de ce dernier dans les 17 applications en effectuant une identification binaire pour détecter si le patron est appliqué ou non dans chaque application.

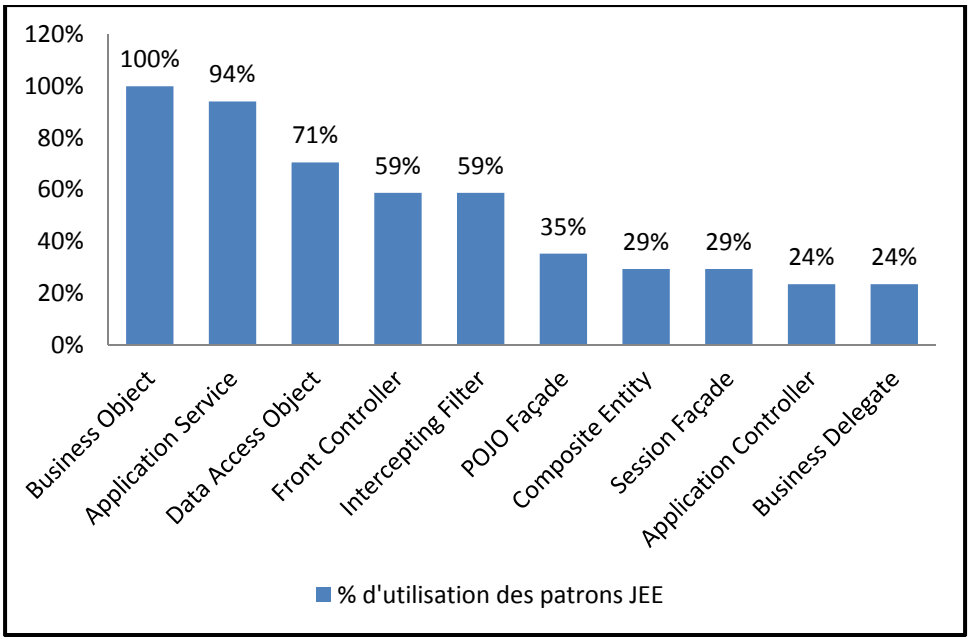

Figure 4.1 Pourcentage d'utilisation de dix patrons JEE dans 17 applications JEE

D'après la Figure 4.1, nous constatons que les dix patrons JEE étudiés ont été appliqués dans les applications JEE sélectionnées, mais le degré de leur utilisation varie d'une application à l'autre.

« *Business Object* », « *Application Service* » et « *DAO* » sont les patrons les plus utilisés :

- « *Business Object* » est utilisé dans toutes les applications. En effet, il est évident d'identifier l'existence de ce patron dans toutes les applications, car il représente les objets métiers, qui sont des concepts incontournables dans chaque application.
- « *Application Service* » centralise la logique d'affaires qui agit sur plusieurs « *Business Object* ». Il peut correspondre à un cas d'utilisation. Ce qui explique son large emploi.
- « *DAO* » comprend les fonctionnalités relatives à l'accès à la base de données. Bien que ces fonctionnalités puissent être incluses dans d'autres composants qui décrivent les traitements, l'utilisation fréquente de ce patron montre bien la bonne séparation entre la logique métier et la logique de persistance.

« *Intercepting Filter* » et « *Front Controller* » sont les patrons moyennement utilisés :

- Les deux patrons résolvent des problèmes reliés au contrôle des requêtes du client :
  - « *Intercepting Filter* » permet d'implémenter les prétraitements et les post-traitements des requêtes clients. Il permet aussi de les décorer.
  - « *Front Controller* » permet de traiter les requêtes et générer les réponses. Il est généralement implémenté comme une *servlet*, qui n'est pas toujours le cas. Ce patron peut être implémenté comme une page JSP ou bien ses traitements sont inclus dans l'un des patrons de la couche présentation, qui est responsable du contrôle des requêtes. Ce qui explique la raison pour laquelle ce patron est modérément utilisé.
- Les deux patrons sont complémentaires et ils peuvent être utilisés ensemble dans les applications (6 applications parmi les 17 applications analysées ont utilisé ces deux patrons ensemble).

« *Session Facade* » et « *POJO Facade* » ensemble sont moyennement utilisés (ensemble ils ont un pourcentage de 64%) :

- Les deux représentent une façade pour la couche métier. La différence entre les deux est juste en implémentation (le premier est une simple classe Java (POJO), tandis que le deuxième est un composant EJB). Ces deux patrons jouent en effet le même rôle, mais ils utilisent deux différents types de composant pour l'implémentation.

« *Composite Entity* », « *Application Controller* » et « *Business Delegate* » sont les patrons les moins utilisés:

- « *Composite Entity* » permet de représenter les composants comme étant une hiérarchie d'objets persistants. Son utilisation dépend de la complexité de l'application et de l'utilisation ou non du modèle de composants EJB.
- « *Application Controller* » centralise la gestion des commandes et des vues dans une application. Il est très peu utilisé. La non-utilisation flagrante de ce patron est due au fait que les traitements de gestion peuvent ne pas être centralisés dans un élément unique, mais plutôt inclus dans d'autres patrons.

- « *Business Delegate* » représente le point de sortie de la couche présentation pour accéder aux composants métiers. Il est très peu utilisé ce qui signifie que l'accès à la couche métier se fait directement sans l'intervention d'un intermédiaire, (i.e., les entités de la couche présentation appellent directement la façade de la couche métier).

4.1.2 Cooccurrences des patrons dans les applications

La Figure 4.2 présente la distribution des patrons étudiés dans chaque application. Les applications sont ordonnées en fonction de leur nombre de classes.

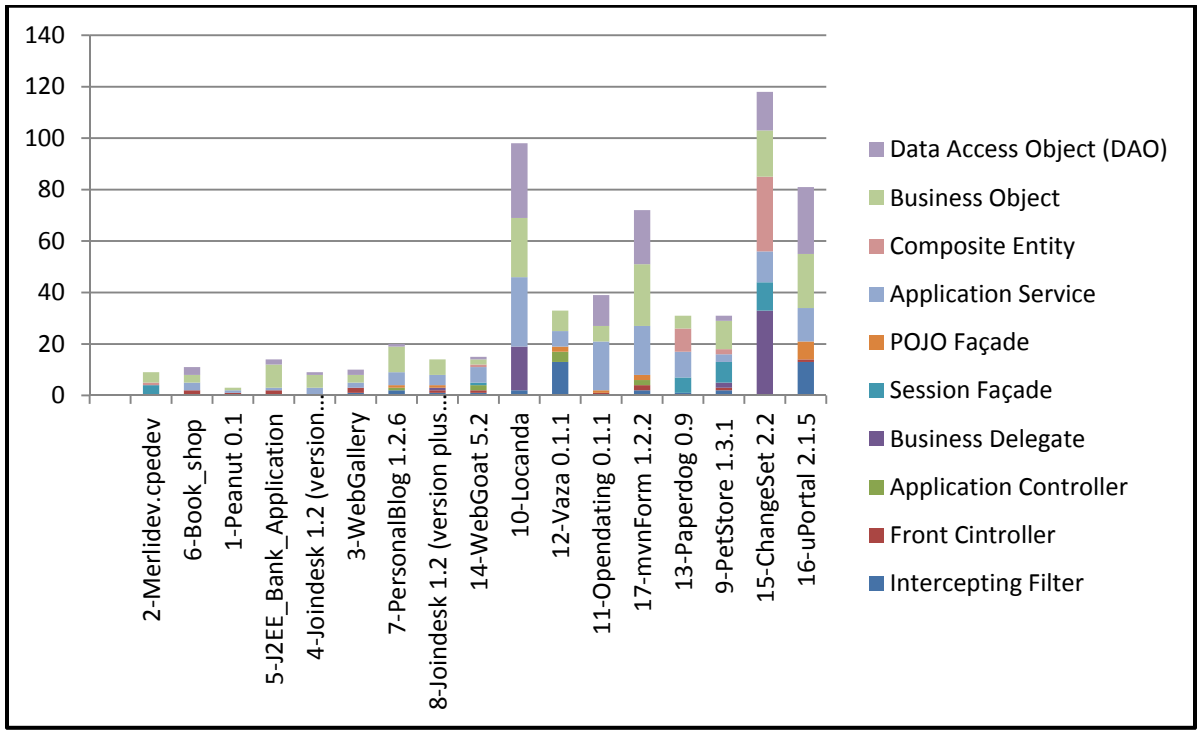

Figure 4.2 Nombre d'occurrences des patrons JEE détectés et leur réparation dans chaque application

En se basant sur les résultats présentés dans la Figure 4.2, il est clair qu'aucune application parmi les 17 applications étudiées n'utilise, à la fois, les neuf patrons JEE à détecter. Nous rappelons que les deux patrons (« *POJO Facade* » *et* « *Session Facade* ») jouent le même rôle.

Une application JEE est généralement conçue en combinant plusieurs patrons. En effet, ces patrons sont complémentaires puisqu'ils permettent de gérer différents problèmes de conception. Prenant le cas du patron « *Application Controller* » qui reçoit la demande de la part d'un « *Font Controller* » ou d'un « *Intercepting Filter* » et qui utilise un de ces quatre patrons (« *Business Delegate* », « *Application Service* », « *POJO Facade* » ou « *Session Facade* ») pour accéder aux traitements de l'application. Nous avons constaté que ces dépendances existent bien dans les applications analysées. De plus, nous constatons que les trois patrons de la couche présentation (« *Front Controller* », « *Intercepting Filter* » et « *Application Controller* »), qui sont responsable des contrôles des requêtes client, sont appliqués ensemble dans les grandes applications. Cela indique une bonne manière d'application de ces patrons.

Il est à noter aussi que le nombre de patrons utilisés est proportionnel à la taille et la complexité de l'application. Plus l'application est grande, plus le nombre de patrons appliqués est important. La croissance du nombre de patrons appliqués est bien visible dans l'histogramme de la Figure 4.2 sachant que les applications sont ordonnées en fonction de leur nombre de classes. L'application '*Locanda*' constitue une exception, car elle ne suit pas la même pente de croissance. En effectuant l'analyse de cette application, nous constatons que l'implémentation est faite de façon fragmentée. Cette application utilise un nombre important d'objets persistants « *Business Object* », dont chacun possède son propre « DAO » et son propre « *Application Service* » qui implémente respectivement les fonctionnalités relatives à l'accès à la base de données et la logique d'affaires qui agissent sur cet objet. Pour cela, cette application utilise un nombre important de patrons JEE.

De plus, Alur *et al.* (2003) ont précisé pour certains patrons le nombre d'occurrences qui peut être utilisé dans l'application. Par exemple, pour le cas du patron « *Front Controller* », ils ont mentionné que généralement l'application contient une seule occurrence de ce patron afin de traiter toutes les requêtes mais, dans certains cas, il existe plus qu'un « *Front Controller* » dont chacun est spécifique à un ensemble de services. En effet, la majorité des applications analysées utilise un seul « *Front Controller* » ou deux au maximum.

En analysant les patrons détectés dans chaque application, nous constatons que la combinaison la plus utilisée est le triplet *(*« *Application Service* », « *Business Object* » et « DAO »*)*. En effet, il est naturel que ce triplet soit le plus appliqué, car l'utilisation de ces trois patrons ensemble permet de décrire le contenu d'une application (les traitements, les objets métier et les fonctionnalités relatives à l'accès à la base de données).

Nous notons également qu'il existe une relation entre les nombres d'occurrences de patrons appliqués. Par exemple, lorsque le nombre d'occurrences de « *Session Facade* », de « *POJO Facade* », d' « *Application Service* » ou de « *DAO* » est important, le nombre d'occurrences de « *Business Object* » ou de « *Composite Entity* » est élevé. C'est le cas des trois plus grandes applications. Ce nombre d'occurrences est lié à la taille de l'application. En effet, plus l'application est grande, plus le nombre d'occurrences de patrons est important. En particulier, c'est le cas du patron « *Application Service* ». Le nombre d'occurrences de ce patron est important dans les grandes applications.

Théoriquement**,** selon Alur *et al.* (2003) la relation établie entre le patron « *Business Delegate* » et le patron « *Session Facade* » est de cardinalité 1-à-1. Ceci n'est pas le cas dans les applications étudiées, car le nombre d'occurrences de « *Session Facade* » n'est pas égal à celui de « *Business Delegate* ».

4.1.3 Distribution des patrons par domaine

Nous constatons que la distribution des patrons n'est pas la même dans tous les domaines des applications. Ceci vient du fait que certains aspects sont plus significatifs dans un domaine que dans un autre.

Le « *Front Controller* **»** est plus utilisé dans les *Frameworks* (‘*peanut*’, ‘*Joindesk*’ et ‘*uPortal*’) et les applications Web (‘*WebGallery*’, ‘*JEE_Bank_Application*’, ‘*WebGoat*’ et ‘*ChangeSet*’). Nous expliquons l'existence de ce patron dans ces deux domaines par le fait que ce patron est un *servlet*. Ce type de composant est utilisé dans les applications Web

comme l'élément qui manipule les requêtes client reçues par l'intermédiaire du serveur HTTP. Les *Frameworks*, dans lesquels ce patron est appliqué, sont utilisés dans des applications Web.

Le patron « *Intercepting Filter* » est très utilisé dans les applications de domaine Communication ('*Personalblog*', '*Opendating*', '*Vaza*' et '*mvnForm*') et Business ('*PetStore*', '*Locanda*' et '*Paperdog*'). Cela peut être dû au fait que ce patron introduit le contrôle de sécurité des requêtes clients, qui est nécessaire à ces deux domaines. En particulier, les transactions effectuées dans les applications du domaine 'Business' nécessitent plus de sécurité, ce qui peut expliquer l'utilisation de ce patron.

Le patron « *Application Controller* » est utilisé dans le domaine Communication. Cela peut s'expliquer par le fait que ces applications offrent des interfaces composées de plusieurs vues.

Pour les autres patrons, il n'y a pas de distribution significative par domaine.

4.2 Quel est l'impact de l'application de ces patrons JEE sur la modifiabilité des applications ?

Afin d'évaluer l'impact des patrons détectés sur la qualité des applications, plus particulièrement la modificabilité, nous étudions le degré d'association entre (i) l'utilisation des patrons JEE et (ii) la qualité des applications en termes de modifiabilité. De ce fait, nous vérifions s'il existe une corrélation entre (i) et (ii). Tout d'abord, nous commençons par présenter le coefficient de corrélation que nous utiliserons. Par la suite, nous présentons les résultats du calcul de corrélation entre les occurrences des patrons et la modifiabilité. Enfin, nous discutons et interprétons ces résultats ainsi que les résultats de l'évolution de la distribution des patrons appliqués dans les différentes versions d'applications analysées en fonction des valeurs des métriques.

4.2.1 Choix du coefficient de corrélation

La corrélation entre deux variables exprime la force de l'association entre ces variables. Les coefficients de corrélation les plus connus sont le coefficient de corrélation de Spearman et celui de Pearson. Gautheir a rapporté la forte ressemblance entre ces deux coefficients. La normalité de la distribution est la contrainte qui permet de distinguer entre l'utilisation de ces deux coefficients. Selon (Bosquet) et (Gautheir, 2001), si la distribution des données suit la loi normale, le meilleur coefficient de corrélation à utiliser est le coefficient de Pearson. Donc, une vérification de la normalité de la distribution est nécessaire pour décider quel est le meilleur coefficient à utiliser dans notre étude. La démarche suivie pour justifier le choix du coefficient de corrélation à utiliser est illustrée dans la Figure 4.3.

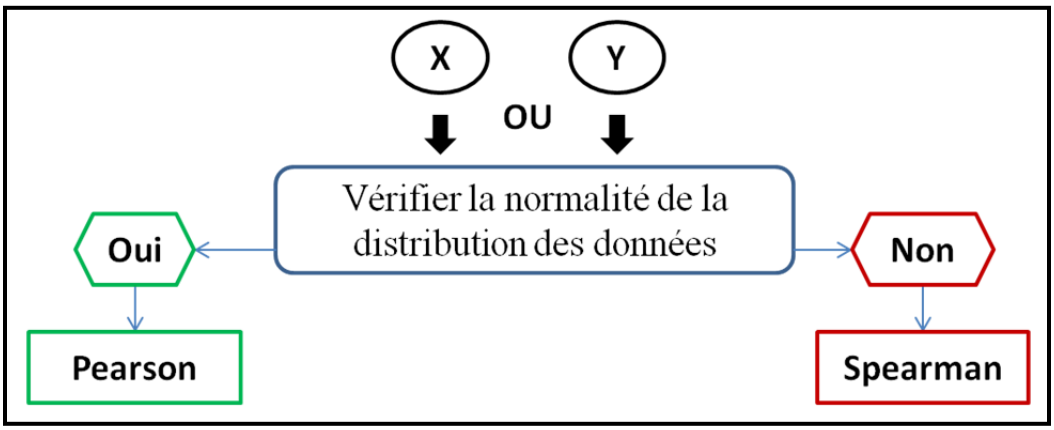

Figure 4.3 Démarche suivie pour le choix du coefficient de corrélation
Adaptée de Bosquet, Laurent[39].

Dans notre étude, les deux variables pour lesquelles le coefficient de corrélation est calculé sont les suivantes: (i) les données recueillies sur les mesures, qui sont présentées dans la section 3.1.2 et (ii) les données sur les patrons détectés, qui sont présentées dans la section 3.2.2 (tableaux 3.1 et 3.4).

[39] Bosquet, Laurent. « Méthodologie de la recherche ».

Rakotomalala (2008) a présenté plusieurs méthodes et techniques pour vérifier la conformité d'une variable à la loi normale. Nous optons pour l'utilisation de la méthode graphique au moyen de l'histogramme de fréquence. Rakotomalala a défini cette méthode comme étant la plus simple à utiliser. Celle-ci consiste à « couper automatiquement l'intervalle de définition de la variable en k (k=$Log_2(n)$, avec n : nombre d'échantillons) intervalles de largeur égaux, puis de produire une série de barres dont la hauteur est proportionnelle à l'effectif associé à l'intervalle. » (Rakotomalala, 2008, p.5). Nous avons utilisé le complément *Analysis ToolPak* du logiciel *Excel* afin de présenter nos données sous la forme d'histogrammes de fréquences.

Les résultats complets de l'analyse de distribution des données pour l'ensemble des métriques sont présentés dans l'ANNEXE II. Nous constatons qu'aucune des métriques n'a une distribution normale. Donc, le coefficient de corrélation le plus adéquat à notre étude est celui de Spearman. Selon (Gautheir, 2001), (Hollander, Wolfe et Chicken, 2013) et (Bosquet), ce coefficient n'est pas basé sur les valeurs réelles de deux variables, mais sur le rang de chaque variable. La valeur de ce coefficient varie entre -1 et 1.

Pathak (2011) distingue deux cas de calcul du coefficient de corrélation. Le cas le plus simple est lorsque les données ne contiennent aucune redondance, alors que le cas complexe est lorsqu'il y a des données avec des rangs équivalents. La différence entre les deux cas se résume lors de l'affectation des rangs aux données. Selon (Pathak, 2011), la formule (4.1) est utilisée pour calculer ce coefficient.

$$r_s = 1 - \frac{6\sum_{i=1}^{n} d_i}{n^3 - n} \tag{4.1}$$

Tirée de Pathak (2011, p.65)

où n est définie par la taille de l'échantillon qui est dans notre cas égale à 17, car nous travaillons sur 17 applications et d est la différence entre les rangs des deux valeurs.

Afin d'évaluer le degré de signification des valeurs de coefficient de corrélation calculées, nous calculons la valeur p. La valeur p est utilisée dans les tests statistiques afin de justifier

les conclusions tirées des résultats obtenus. Les conclusions dépendent de deux hypothèses qui sont :

- H_0 (l'hypothèse nulle) qui stipule l'inexistence de relation entre les deux variables
- H_1 (l'hypothèse alternative) qui stipule l'existence d'une relation entre les deux variables).

Le calcul de la valeur p se fait en référant à un seuil bien défini (seuil= 5 % = 0.05). Deux cas de figure se présentent :

- Si la valeur p < seuil, on rejette l'hypothèse nulle en faveur de l'hypothèse alternative.
- Si la valeur p > seuil, on ne rejette pas l'hypothèse nulle et on ne peut rien conclure.

Pour les cas où la valeur p < seuil, nous notons que :

- plus la valeur de r_s est proche de -1, plus la corrélation est forte. Dans ce cas, on dit que la corrélation est négative (c.-à-d. lorsque x (1re entrée) augmente, y (2e entrée) diminue).
- plus la valeur de r_s est proche de 1, plus la corrélation est forte. Dans ce cas, on dit que la corrélation est positive (c.-à-d. lorsque x (1re entrée) augmente, y (2e entrée) augmente aussi).
- Lorsque la valeur de r_s est proche de 0, aucune corrélation n'existe entre les deux variables x (1re entrée) et y (2e entrée).

4.2.2 Normalisation des occurrences des patrons par la taille de l'application

Le nombre de patrons détectés est plus important dans les grandes applications. De ce fait, nous constatons que la taille influence le nombre de patrons détectés. Par conséquent, pour minimiser l'impact de la taille sur le nombre des patrons détectés, nous décidons de normaliser le nombre de patrons détectés en fonction du nombre de classes. En premier lieu,

nous normalisons le nombre de classes (c.-à.-d. NOC) en utilisant la formule standard de la normalisation suivante[40] (4.2):

$$\text{Val}_{\text{normalisé}} = \frac{(Val - \min) * (\text{MAX} - \text{MIN})}{(max - \min)} + \min \qquad (4.2)$$

Tirée de Hardwre.fr (2008)

où : $\text{Val}_{\text{normalisé}}$ est la valeur normalisée

Val est la valeur originale

[min, max] est l'intervalle original (c.-à.-d. min = la valeur minimale de l'ensemble des valeurs et max = la valeur maximale de l'ensemble des valeurs)

[MIN, MAX] est l'intervalle cible

Comme le coefficient de Spearman se base sur les rangs, nous pouvons choisir un intervalle pour les rangs déduits des valeurs mesurées. Cet intervalle n'aura pas d'impact sur les résultats de corrélations. Dans notre cas, nous choisissons comme intervalle cible [1, 10], donc les valeurs normalisées de la mesure NOC seront des valeurs de l'intervalle [1, 10]. Le résultat de la normalisation est illustré dans le Tableau 4.1.

[40] HARDWARE.FR. 1997. En ligne <http://forum.hardware.fr/hfr/Programmation/Algo/normalisation-valeurs-sujet_116966_1.htm>. Consulté le 1 avril 2015.

Tableau 4.1 Résultat de la normalisation du nombre de classes NOC

NOC normalisé= ((NOC - min) * 9 / (max - min))+1

	NOC	NOC normalisé
1-**Peanut** 0.1	18	1,09
2-**Merlidev.cpedev**	13	1
3-**WebGallery**	47	1,63
4-**Joindesk** 1.2 (version minimale)	37	1,44
5-**J2EE_Bank_Application**	26	1,24
6-**Book_shop**	16	1,06
7-**PersonalBlog** 1.2.6	47	1,63
8-**Joindesk** 1.2 (version plus complète)	137	3,3
9-**PetStore** 1.3.1	309	6,48
10-**Locanda**	152	3,57
11-**Opendating** 0.1.1	246	5,31
12-**Vaza** 0.1.1	160	3,72
13-**Paperdog** 0.9	299	6,3
14-**WebGoat** 5.2	145	3,44
15-**ChangeSet** 2.2	406	8,28
16-**uPortal** 2.1.5	499	10
17-**mvnForm** 1.2.2	273	5,81
min	13	
max	499	

Après la normalisation du nombre de classes, l'étape suivante consiste à normaliser les nombres de patrons détectés dans chaque application (Tableau 3.1) en les divisant par le $NOC_{normalisé}$ qui convient. Le résultat de la normalisation du nombre d'occurrences des patrons dans les 17 applications est illustré dans le Tableau 4.2.

Tableau 4.2 Nombres de patrons JEE normalisés dans chaque application

	Front Controller	Data Access Object (DAO)	Business Delegate	Intercepting Filter	Application Controller	Composite Entity	Business Object	Application Service	Session Façade	POJO Façade	Façade (Session & POJO)
1-Peanut 0.1	0,91743119	0	0	0	0	0	0,91743119	0,91743119	0	0	0
2-Merlidev.cpedv	0	0	0	0	0	1	4	0	4	0	4
3-WebGallery	1,22699387	1,226993865	0	0,61349693	0	0	1,8404908	1,22699387	0	0	0
4-Joindesk 1.2 (version minimale)	0	0,694444444	0	0	0	0	3,47222222	2,08333333	0	0	0
5-J2EE_Bank_Application	1,61290323	1,612903226	0	0	0	0	7,25806452	0,80645161	0	0	0
6-Book_shop	1,88679245	2,830188679	0	0	0	0	2,83018868	2,83018868	0	0	0
7-PersonalBlog 1.2.6	0	0,591715976	0	1,18343195	0,59171598	0	5,91715976	2,95857988	0	0,59172	0,591715976
8-Joindesk 1.2 (version plus complète)	0,3030303	0	0,3030303	0,3030303	0	0	1,81818182	1,21212121	0	0,30303	0,303030303
9-PetStore 1.3.2	0,15432099	0,308641975	0,30864198	0,30864198	0	0,30864198	1,69753086	0,46296296	1,23457	0	1,234567901
10-Locanda	0	8,1232493	4,76190476	0,56022409	0	0	6,44257703	7,56302521	0	0	0
11-Opendating 0.1.1	0,18832392	2,259887006	0	0	0	0	1,1299435	3,57815443	0	0,18832	0
12-Vaza 0.1.1	0	0	0	3,49462366	1,07526882	0	2,15053763	1,61290323	0	0,53763	0,537634409
13-Paperdog 0.9	0	0	0	0,15873016	0	1,42857143	0,79365079	1,58730159	0,95238	0	0,952380952
14-WebGoat 5.2	0,29069767	0,290697674	0	0,29069767	0,58139535	0,29069767	0,58139535	1,74418605	0,2907	0	0,290697674
15-ChangeSet 2.2	0	1,811594203	3,98550725	0	0	3,50241546	2,17391304	1,44927536	1,3285	0	1,328502415
16-uPortal 2.1.5	0,1	2,6	0	1,3	0	0	2,1	1,3	0	0,7	0,7
17-mvnForm 1.2.2	0,34423408	3,614457831	0	0,34423408	0,34423408	0	4,13080895	3,27022375	0	0,34423	0,344234079

4.2.3 Résultats du calcul de coefficient de Spearman

Vue que le nombre d'applications (17), celui de patrons détectés (dix patrons JEE) et celui de métriques (19 métriques) sont importants, nous avons identifié un outil qui nous permet de faciliter le calcul. Cet outil s'appelle "*Statistical tools for high-throughput data analysis*" (**STHDA**[41]). Il s'agit d'une application web qui permet de calculer les coefficients de corrélation de Pearson, de Kendall et de Spearman et aussi la valeur p. Il prend comme entrée les listes de valeurs de chaque variable. Dans notre cas, ces deux listes sont : (i) la liste du nombre d'occurrences des patrons normalisés et (ii) la liste des valeurs des métriques. L'application fournit en sortie la valeur du coefficient de corrélation de Spearman et la valeur p. Les résultats de calcul du coefficient de corrélation de Spearman et de la valeur p sont respectivement présentés dans les Tableaux 4.3 et 4.4.

Le Tableau 4.3 présente les coefficients de corrélation de Spearman r_s pour toutes les paires patron JEE – métrique. Dans le tableau mentionné, nous avons présenté en vert les valeurs de r_s dont le niveau de signification (valeur p) inférieur ou égal à 5 % (les données complètes du calcul de valeur p est présenté dans le Tableau 4.4). Les résultats indiquent une forte corrélation entre 11 paires patron JEE – métrique. Toutes ces corrélations sont des corrélations positives.

[41] STHDA. Logiciel. En ligne. <http://www.sthda.com/rsthda/correlation.php>.

Tableau 4.3 Coefficient de corrélation de Spearman entre la fréquence des patrons JEE et les métriques

	Taille					Complexité		Couplage						Cohésion		Héritage
	NOM	NOC	NOP	NA	LOC	WMC	CYCLO	CBO	RFC	ACAIC	ACMIC	CA	CE	LCOM1	LCOM2	DIT
Front Controller	-0.4206	-0.3316	-0.2724	-0.3606	-0.3276	-0.3708	-0.3429	-0.287	-0.2362	-0.1194	-0.4743	-0.0178	-0.2667	-0.2565	-0.3022	-0.0657
Session Façade	0.0806	0.2161	0.3415	0.2357	0.1688	0.1749	0.1886	0.593	0.2098	0.237	-0.1653	0.1232	0.187	-0.0684	-0.0715	-0.2938
DAO	0.2682	0.2309	-0.0249	0.0645	0.263	0.2184	0.201	0.0447	0.072	0.2434	0.07	0.4492	0.2556	0.3499	0.3226	0.1871
Business Delegate	0.3097	0.3757	0.3499	0.2141	0.2404	0.2371	0.2338	0.1616	0.0659	0.2965	0.1127	0.0494	0.3063	-0.1416	-0.1449	-0.1386
Intercepting Filter	0.6429	0.4231	0.3195	0.5105	0.4978	0.5079	0.4698	-0.1256	0.3987	-0.0921	0.5331	0.2895	0.3632	0.2971	0.3124	-0.2311
Application Controller	0.3196	0.084	-0.0512	0.3096	0.3655	0.2865	0.2766	0.1291	0.5335	0.0088	0.1724	0.1383	0.112	0.3227	0.3557	-0.0497
Composite Entity	0.1202	0.2982	0.4085	0.2874	0.2418	0.26	0.2768	0.458	0.146	0.2288	-0.0919	0.1688	0.2661	-0.1627	-0.1566	-0.2295
Business Object	-0.0736	-0.2869	-0.4349	-0.2941	-0.1887	-0.1789	-0.2108	-0.3317	0.2451	0.1482	-0.0025	-0.2279	-0.2819	0.4828	0.4877	0.3027
Application Service	0.3532	0.179	-0.043	0.1691	0.3358	0.2819	0.2745	0.0302	-0.0196	0.0461	0.3529	0.3627	0.2034	0.3358	0.3652	0.3677
POJO Façade	0.4649	0.3558	0.2041	0.4675	0.4388	0.4388	0.3872	-0.0134	0.2438	-0.0848	0.3783	0.1864	0.3786	0.1749	0.2208	0.0505
Façade (Session & POJO)	0.399	0.46	0.4748	0.5536	0.4419	0.4851	0.4521	0.2857	0.4089	0.2491	0.1649	0.1448	0.4292	0.0508	0.0787	0.0581

Tableau 4.4 Les valeurs de Valeur-P

	Taille					Complexité		Couplage						Cohésion		Héritage
	NOM	NOC	NOP	NA	LOC	WMC	CYCLO	CBO	RFC	ACAIC	AMAIC	CA	CE	LCOM1	LCOM2	DIT
Front Controller	0.09276	0.1935	0.2901	0.155	0.1992	0.1429	0.1779	0.2641	0.3614	0.648	0.0544	0.946	0.3008	0.3203	0.2384	0.8022
Session Façade	0.7583	0.4049	0.1798	0.3625	0.5172	0.502	0.4686	0.0121	0.4189	0.3597	0.526	0.6377	0.4723	0.7941	0.7852	0.2524
DAO	0.298	0.3725	0.9245	0.8057	0.3077	0.3998	0.4392	0.8648	0.7837	0.3465	0.7896	0.0705	0.3221	0.1686	0.2066	0.4722
Business Delegate	0.2263	0.1373	0.1685	0.4094	0.3527	0.3595	0.3664	0.5355	0.8017	0.2479	0.6666	0.8507	0.2319	0.5877	0.579	0.5958
Intercepting Filter	0.005372	0.0906	0.2113	0.03628	0.04202	0.03738	0.05704	0.6309	0.1129	0.7251	0.02755	0.2597	0.1519	0.2468	0.2222	0.3721
Application Controller	0.2111	0.7485	0.8454	0.2266	0.1491	0.2649	0.2824	0.6215	0.02742	0.9731	0.5082	0.5965	0.6688	0.2064	0.1612	0.8499
Composite Entity	0.6459	0.245	0.1035	0.2634	0.3498	0.3135	0.2822	0.0482	0.5761	0.377	0.7259	0.5172	0.3019	0.5327	0.5483	0.3755
Business Object	0.779	0.2641	0.08106	0.2511	0.4668	0.4907	0.4152	0.1933	0.3417	0.5703	0.9925	0.3774	0.2721	0.05162	0.04904	0.2377
Application Service	0.1644	0.4918	0.8698	0.5152	0.1876	0.2721	0.2852	0.9083	0.9435	0.8605	0.1647	0.1529	0.4321	0.1876	0.15	0.1465
POJO Façade	0.06008	0.161	0.432	0.05848	0.07807	0.07807	0.1247	0.9593	0.3458	0.7463	0.1344	0.4738	0.134	0.5019	0.3944	0.8475
Façade (Session & POJO)	0.1126	0.06321	0.05411	0.02113	0.07574	0.04842	0.06849	0.2664	0.1032	0.335	0.527	0.5793	0.08557	0.8465	0.7639	0.8247

4.2.4 Analyse des corrélations par patron

« *Intercepting Filter* » : Ce patron permet d'implémenter chaque traitement de contrôle, effectué sur les demandes clients, dans un filtre séparé. Plus le nombre d'occurrences de ce patron est important, plus le nombre de classes qui décrivent les prétraitements et les post-traitements effectués sur les demandes augmente. Par conséquent, le nombre de méthodes, celui d'attributs et celui de lignes de code source augmentent. De plus, une croissance de nombre d'occurrences de ce patron va compliquer l'arbre de décisions, donc le chemin suivi en invoquant une méthode va devenir plus long (WMC augmente).

« *Session Facade* » et **« *POJO Facade* » (Façade)**: Ces deux patrons jouent le même rôle malgré qu'ils soient implémentés avec deux types de composants différents. Pour cette raison, nous avons combiné le nombre de « ***Session Facade*** » détecté et le nombre de « ***POJO Facade*** » détecté. Nous avons aussi calculé le coefficient de corrélation de Spearman des nouvelles valeurs obtenues. Le résultat de ces calculs est présenté dans la dernière ligne du Tableau 4.2. Il existe deux corrélations significatives pour ces patrons avec des métriques. Comme l'utilisation de ces patrons introduit de nouvelles classes qui sont référencées par plusieurs autres classes, cela implique l'augmentation du nombre d'attributs. De plus, une croissance du nombre d'occurrences de ce patron va ajouter un nouveau niveau dans l'arbre de décisions et donc le chemin suivi en invoquant une méthode va devenir plus long (WMC augmente). En outre, plus le nombre de façades est important, plus le nombre de classes couplées aux façades augmente (CBO augmente). Ceci est le cas du patron « ***Session Facade*** ».

« *Application Controller* » : Une seule corrélation est observée entre ce patron et RFC. Comme le rôle de ce patron est l'identification de l'opération à exécuter et le contrôle du flux de données envoyé à la vue, le nombre de méthodes exécutées pour répondre à une requête (RFC) augmente.

« *Business Object* » : Nous observons l'existence d'une seule corrélation significative entre ce patron et LCOM2. Ceci peut être expliqué par le fait que la majorité des **« *Business Object* »** correspondant à des objets persistants. Chaque objet définit des attributs et les accesseurs et les mutateurs pour manipuler ces attributs. Ceci augmente le nombre de méthodes dans une classe qui utilisent un sous-ensemble réduit d'attributs.

« *Composite Entity* » : Une seule corrélation est observée entre ce patron et CBO. Comme cette entité est une entité composite qui est composée de **« *Business Object* »**. **« *Composite Entity* »** introduit plus de références aux **« *Business Object* »**. Ceci augmente le nombre de classes couplées dans l'application, ce qui explique la corrélation entre ce patron et CBO. L'absence d'autres corrélations peut être due au fait que ce patron ne peut être qu'un composant EJB et que cette technologie n'est pas très utilisée dans plusieurs des applications étudiées.

« Front Controller », « Business Delegate », « Application Service » et « Data Access Object »: Aucune corrélation n'est trouvée entre ces patrons et les métriques. Nous expliquons l'absence de corrélation significative entre le patron **« *Front Controller* »** et les métriques au fait que nous n'avons aucun contrôle sur les pages JSP pour vérifier si ce patron est appliqué, car ce dernier peut être utilisé comme étant une page JSP. Pour le cas du patron, **« *Business Delegate* »,** l'absence de corrélation peut être expliquée par le fait que dans la majorité des applications étudiées ce patron n'est pas appliqué. Ce qui est le contraire pour le cas du patron **« *Application Service* »**. Même si ce patron est très appliqué dans les applications analysées, aucune corrélation n'est trouvée entre ce dernier et les métriques. Pour le patron **« *Data Access Object* »,** aucune corrélation n'est trouvée entre ce patron et les métriques. Mais en analysant les valeurs-P correspondantes à ce patron, nous constatons que la valeur-P qui correspond à CA, est proche de 0.05. Celui-ci peut être expliqué par le fait que ce patron encapsule les fonctionnalités relatives à l'accès aux données, donc il va être utilisé par les patrons de la couche métier pour accéder à la base de données, ce qui explique l'augmentation de CA.

Toutefois, une paire fortement corrélée parmi celles présentées dans le Tableau 4.3 et l'inexistence de corrélations entre certains patrons JEE et les métriques peuvent être remises en question. Aucun argument ne peut justifier l'existence de cette relation entre (ACMIC – « ***Intercepting Filter*** ») car ce patron n'est couplé à aucun objet, il effectue plutôt des traitements sur les requêtes clients. En plus, les filtres qui décrivent les prétraitements et les post-traitements implémentent l'interface *javax.servlet.Filter* dont les traitements de contrôle des requêtes client sont implémentés dans la méthode *doFilter* (une méthode héritée). Cette méthode possède des paramètres d'entrée prédéfinis et qui ne sont pas des classes de l'application. Cependant, certaines corrélations significatives qui ont été attendues reflétant les rôles des patrons ne sont pas trouvées. Ceci peut être dû au nombre limité d'applications étudiées ainsi qu'au nombre de fausses occurrences trouvées et au nombre d'occurrences positives non trouvées.

4.3 Analyse des résultats pour différentes versions d'applications JEE

Nous rappelons que nous avons analysé différentes versions de trois applications JEE ('*Joindesk*', '*mvnForum*' et '*Java Pet Store*') pour évaluer l'évolution de la distribution des patrons appliqués dans ces versions et d'évaluer l'impact de cette évolution sur les valeurs des métriques. Plus précisément, nous avons analysé quatre versions des deux premières applications ('*Joindesk*' et '*mvnForum*') et deux versions de la dernière ('*Java Pet Store*'). Nous discutons les résultats par application dans ce qui suit.

4.3.1 Application '*Joindesk*'

Les résultats de la détection des patrons ainsi que le calcul des métriques de cette application sont présentés respectivement dans les Tableaux 4.5 et 4.6.

Tableau 4.5 Nombres de patrons JEE identifiés pour chaque version de l'application '*Joindesk*'

Joindesk (Version minimale)	Front Controller	Data Access Object (DAO)	Business Delegate	Intercepting Filter	Application Controller	Composite Entity	Business Object	Application Service	Façade (Session & POJO)
Joindesk 0.9	0	0	0	0	0	0	4	2	0
Joindesk 1.0	0	1	0	0	0	0	5	3	0
Joindesk 1.1	0	1	0	0	0	0	5	3	0
Joindesk 1.2	0	1	0	0	0	0	5	3	0

Tableau 4.6 Valeurs des métriques de quatre versions de l'application '*Joindesk*'

Joindesk	Taille					Complexité		Couplage						Cohésion		Héritage
(Version minimale)	NOM	NOC	NOP	NA	LOC	WMC	CYCLO	CBO	RFC	ACAIC	ACMIC	CA	CE	LCOM1	LCOM2	DIT
Joindesk 0.9	71	14	9	42	592	107	44	1,3333	8,4	0	0	**1,67**	**14**	13,167	3,5833	1,57
Joindesk 1.0	290	36	16	165	2651	616	424	2,666	12,388	0,013	0,413	**1,27**	**46**	46,267	14,493	2,38
Joindesk 1.1	290	36	16	166	2679	624	432	**2,666**	**12,388**	0,013	0,413	1,27	**46**	46,267	14,493	2,38
Joindesk 1.2	293	37	16	167	2721	631	439	**2,621**	**12,189**	0,013	0,4	1,33	**47**	44,56	15,225	2,36

L'analyse du code de l'ensemble de l'application nous a permis de constater que : (i) cette application contient beaucoup de paquetages comparés au nombre de classes et (ii) la subdivision de l'application en paquetages ne reflète pas la décomposition du système en une hiérarchie en couches. En effet, les paquetages liés à la logique métier ne sont pas regroupés dans un paquetage parent.

En nous référant aux valeurs des métriques du Tableau 4.6, nous constatons que le nombre de classes (paquetages) est multiplié par 2.5 (1.8) en passant de la version 0.9 à la version 1.0. Cela explique l'augmentation des valeurs de la majorité des métriques. Alors que l'évolution des valeurs est presque constante entre les versions restantes sauf pour les deux métriques CBO et RFC. Le passage de la version 1.1 à la version 1.2 est réalisé en ajoutant une nouvelle classe '*SessionClassInfoProxy'*. Cet ajout va impliquer l'augmentation de CBO (RFC) de 1 (respectivement de 5). Mais en termes de moyenne les valeurs de CBO et de RFC ont diminué comme illustré par le Tableau 4.6.

En ce qui concerne le résultat de la détection présenté dans le Tableau 4.5, nous notons que trois occurrences de patrons JEE (« DAO », « *Application Service* » et « *Business Object* ») sont introduites dans la version 1.0. Pour les deux versions suivantes (1.1 et 1.2), aucune nouvelle occurrence de patron JEE n'est introduite. Les changements majeurs de l'application '*Joindesk*' ont été faits en passant de la version 0.9 à la version 1.0. La version 0.9 de cette application contient trois paquetages principaux (business, domain et web). Un nouveau paquetage (database) a été introduit dans la version 1.0. Ce nouveau paquetage contient les classes responsables de l'accès à la base de données. L'occurrence du patron « *DAO* » détectée est une classe de ce nouveau paquetage.

Rappelons que CA signifie le nombre de classes (hors d'un paquetage sélectionné) qui dépendent d'une classe du paquetage sélectionné (les points d'entrées de chaque paquetage) et CE signifie le nombre de classes d'un paquetage qui dépendent d'une classe d'un autre paquetage (les points de sorties de chaque paquetage). En nous référant aux valeurs des métriques du Tableau 4.5, nous constatons que la valeur de CA diminue en passant de la version 0.9 à la version 1.0. L'ajout d'une nouvelle occurrence du patron « *DAO* » va augmenter le nombre de points d'entrées pour accéder aux données. Mais en termes de moyen, l'ajout du nouveau paquetage database (qui contient l'occurrence du patron « *DAO* »), va diminuer la valeur de CA. Par ailleurs, l'ajout des nouvelles occurrences des patrons « *Application Service* », « *Business Object* » et « *DAO* » dans différents paquetages business, domain et database, engendre l'augmentation des dépendances entre les paquetages (plus précisément le nombre de points d'entrées à chaque paquetage). Cela explique la croissance de la valeur de CE (présentées en gras dans le Tableau 4.6).

4.3.2 Application '*mvnForum*'

Les résultats de la détection et du calcul des métriques de cette application sont présentés respectivement dans les Tableaux 4.7 et 4.8.

Tableau 4.7 Nombres de patrons JEE identifiés pour chaque version de l'application '*mvnForum*'

mvnForum	Front Controller	Data Access Object (DAO)	Business Delegate	Intercepting Filter	Application Controller	Composite Entity	Business Object	Application Service	Façade (Session & POJO)
1.0	2	23	0	2	2	0	23	7	0
1.1	2	20	0	2	2	0	24	13	2
1.2	2	20	0	2	2	0	24	15	2
1.2.2	2	20	0	2	2	0	24	18	2

Tableau 4.8 Valeurs des métriques de quatre versions de l'application '*mvnForum*'

mvnForum	Taille					Complexité		Couplage						Cohésion		Héritage
	NOM	NOC	NOP	NA	LOC	WMC	CYCLO	CBO	RFC	ACAIC	ACMIC	CA	CE	LCOM1	LCOM2	DIT
1.0	2917	219	19	1286	43451	7592	3503	**5,95**	**15,89**	0,0103	0,0793	10,69	194	268,21	124,52	1,93
1.1	3773	266	26	1872	48076	9289	4503	**5,47**	**16,44**	0,0196	0,0868	11,3	219	599,95	290,35	1,94
1.2	**3791**	**273**	26	1883	48283	9286	4544	**5,62**	**15,96**	0,0193	0,0856	11,35	222	612,35	296,63	1,92
1.2.2	**3786**	**273**	26	1884	48411	9291	4554	**5,45**	**15,69**	0,0193	0,0856	13,79	222	615,83	298,38	1,92

L'analyse du code de l'application nous a permis de constater que la subdivision de l'application en paquetages ne suit pas une bonne décomposition du système en une hiérarchie en couches. En effet, les entités « *Business Object* » et les « *DAOs* » sont dans le même paquetage (db). De plus, les classes qui décrivent les traitements de l'application (« *Application Service* ») sont situées dans plusieurs sous paquetages du paquetage racine et le partitionnement de l'application n'est pas basé sur le contenu des différentes couches, mais plutôt cette subdivision a une granularité très fine (c.-à.-d. tous les paquetages sont placés dans le paquetage racine de l'application sans utiliser aucun regroupement qui reflète la notion de la décomposition en couches).

En nous référant au Tableau 4.7 qui illustre l'évolution du nombre d'occurrences des patrons, nous observons que de nouvelles occurrences de patrons (deux occurrences du patron « *Façade* », six occurrences du patron « *Application Service* » et une occurrence du patron « *Business Object* ») ont été introduites alors qu'une occurrence du patron « *DAO* » a été éliminée en passant de la version 1.0 à la version 1.1 de l'application '*mvnForum*'. Pour les

autres versions, le seul changement introduit en passant de la version 1.1 à la version 1.2 (respectivement de la version 1.2 à la version 1.2.2) est l'ajout de deux (trois) occurrences du patron « *Application Service* ».

En fait, les changements majeurs se sont faits en passant de la version 1.0 à 1.1. En effet, trois nouveaux paquetages ont été ajoutés (categorytree, event et service) et le paquetage impl (existant dans la version 1.0) a été intégré dans le paquetage service. De plus, plusieurs classes ont été ajoutées et d'autres ont été enlevées ou renommées. Alors que pour les autres versions, des modifications mineures ont été introduites.

Pour ce qui est de l'évolution des métriques, nous constatons à partir du Tableau 4.8, que la valeur de RFC augmente en passant de la version 1.0 à la version 1.1. Cela s'explique par l'introduction de nouvelles occurrences du patron « *Application Service* » et l'introduction de deux nouvelles « *Façades* ». Ces deux patrons introduisent une indirection qui augmente le nombre de méthodes invoquées lors du traitement d'une requête client.

En passant de la version 1.1 à la version 1.2, sept nouvelles classes ont été introduites dont deux d'entre elles sont des occurrences du patron « *Application Service* ». De plus, nous remarquons que les valeurs de RFC de 20 % des classes (51 classes) ont été augmentées. Ces changements ont entrainé une augmentation de la somme totale de RFC, mais en moyenne cette valeur a diminué. C'est le cas aussi pour le passage de la version 1.2 à la version 1.2.2. En analysant ces deux versions, nous avons observé la suppression et l'ajout de plusieurs classes avec le même nombre total de classes, mais un moindre nombre total de méthodes. Ces changements ont influencé les valeurs de RFC de 23 classes. Ces valeurs ont augmenté (par exemple, la valeur de RFC de la classe '*WatchBean'* passe de 29 (version 1.2) à 33 (version 1.2.2)) ou diminué (par exemple, la valeur de RFC de la classe '*SearchService'* passe de 22 (version 1.2) à 18 (version 1.2.2)). En plus, deux classes qui possèdent une valeur importante de RFC ont été éliminées dans la version 1.2.2. Ces classes sont '*MemberDAOExternalUserDatabaseAbstract'* (RFC=83) et '*MyUtil'* (RFC=34). Toutes ces

modifications ont causé cette diminution du RFC en passant de la version 1.2 à la version 1.2.2.

Par ailleurs, nous constatons que la valeur de CBO diminue en passant de la version 1.0 à la version 1.1. Cela s'explique par l'introduction de nouvelles occurrences du patron « *POJO Facade* » qui va diminuer le nombre total de classes couplées. En plus, cette diminution est expliquée par la réduction de CBO de 31 classes dans la version 1.1 dont certains entre eux sont des occurrences des patrons « *Application Service* » et « *Business Object* ». Par contre, le passage de la version 1.1 à la version 1.2 a engendré l'augmentation de CBO. Cela s'explique par l'introduction de nouvelles occurrences du patron « *Application Service* » qui va augmenter le nombre de classes couplées qui sont des occurrences des « *Business Object* ». Ce qui est le cas, car le CBO de 44 classes ont été augmenté dont certains entre elles sont des occurrences du patron « *Business Object* ».

En passant de la version 1.2 à la version 1.2.2, la valeur de CBO est diminuée. Cela s'explique principalement par la réduction de la valeur de CBO de 42 classes de la version 1.2.2.

4.3.3 Application '*Java Pet Store*'

Les résultats de la détection des patrons ainsi que le calcul des métriques de cette application sont présentés respectivement dans les Tableaux 4.9 et 4.10.

Tableau 4.9 Nombres de patrons JEE identifiés pour chaque version de l'application '*Java Pet Store*'

Java Pet Store	Front Controller	Data Access Object (DAO)	Business Delegate	Intercepting Filter	Application Controller	Composite Entity	Business Object	Application Service	Façade (Session & POJO)
1.1.2	1	8	2	0	0	0	9	2	3
1.3.1	1	5	2	2	0	2	11	3	8

Tableau 4.10 Valeurs des métriques de deux versions de l'application '*Java Pet Store*'

Java Pet Store	Taille					Complexité		Couplage						Cohésion		Héritage
	NOM	NOC	NOP	NA	LOC	WMC	CYCLO	CBO	RFC	ACAIC	ACMIC	CA	CE	LCOM1	LCOM2	DIT
1.1.2	864	233	136	410	12219	2246	986	**3,45**	**7,24**	0	0,0092	**5,24**	179	18,581	6,111	2,38
1.3.1	1696	309	310	1103	17299	3538	1798	**3,58**	**12,4**	0,015	0,0532	**3,32**	219	28,064	8,894	1

L'analyse du code de l'ensemble de l'application nous a permis de constater que la subdivision de l'application en paquetages a beaucoup changé en passant de la version 1.1.2 à la version 1.3.1. En effet, de nouvelles fonctionnalités ont été ajoutées et d'autres classes ont été renommées ou déplacées. De plus, le partitionnement de l'application n'est pas basé sur le contenu des différentes couches, mais de façon que chaque sous paquetage qui décrit un service spécifique est composé d'autres sous-paquetages, citons par exemple, les sous-paquetages : ejb qui contient les « *Sessions Facades* », dao qui contient les occurrences du patron « DAO », model qui contient les « *Composite Entity* » et les « *Business Object* ».

En nous référant au Tableau 4.9 qui illustre l'évolution du nombre d'occurrences des patrons, nous observons que 12 occurrences de patrons JEE sont introduites dans la version 1.3.1, dont deux occurrences du patron « *Intercepting Filter* », deux occurrences du patron « *Composite Entity* », deux occurrences du patron « *Business Object* », une occurrence du patron « *Application Service* » et cinq occurrences du patron « *Session Facade* ». De plus, trois occurrences du patron « *DAO* » ont été éliminées.

L'analyse du code de l'ensemble de l'application nous a permis de constater que plusieurs changements ont été faits à la version 1.3.1. En effet, 16 nouveaux paquetages ont été ajoutés dans le paquetage racine, citons par exemple les paquetages address, cart et catalogue. De plus, plusieurs classes ont été ajoutées et d'autres ont été éliminées ou renommées.

Pour ce qui est de l'évolution des métriques, nous constatons, à partir du Tableau 4.10, que la valeur de RFC augmente en passant de la version 1.1.2 à la version 1.3.1. Cela s'explique par l'introduction de nouvelles occurrences des patrons « *Application Service* », « *Composite*

Entity » et « *Session Facade* ». Ces patrons ont introduit une indirection qui augmente le nombre de méthodes invoquées lors du traitement. Par ailleurs, nous observons que la valeur de CBO est augmentée légèrement en passant de la version 1.1.2 à la version 1.3.1 même si le nombre de classes NOC est doublé. Cela s'explique par le fait que les classes ajoutées décrivent de nouvelles fonctionnalités et elles n'ont pas augmenté le CBO des classes existantes.

En nous référant aux valeurs des métriques du Tableau 4.10, nous constatons que la valeur de CA diminue en passant de la version 1.1.2 à la version 1.3.1. L'élimination de trois occurrences du patron « *DAO* » va diminuer le nombre de points d'entrés pour accéder aux données, alors que l'ajout de cinq occurrences du patron « *Session Facade* » va augmenter le nombre de points d'entrés à la couche métier. Mais en termes de moyen, l'ajout de plusieurs paquetages à la version 1.3.1 ainsi que le déplacement de plusieurs classes ont diminué la valeur de CA.

4.4 Conclusion

Nous avons présenté dans ce chapitre la dernière phase de l'étude empirique. Nous avons analysé et interprété les résultats collectés dans le chapitre précédent. Dans le prochain chapitre, nous résumerons les résultats déjà présentés et nous présenterons les limites de l'étude empirique.

CHAPITRE 5

SYNTHÈSE DES RÉSULTATS ET LIMITES DE L'ÉTUDE

Dans ce chapitre, nous résumons les résultats discutés dans le chapitre précédent. En particulier, nous synthétisons ces résultats en fonction des questions de recherche visées par l'étude. Nous discutons aussi les limites de notre étude.

5.1 Synthèse des résultats

Nous rappelons que notre étude avait pour objectif de répondre aux deux questions suivantes de recherche :

- (Q1) : Jusqu'à quel degré les patrons JEE qui supportent la modifiabilité sont-ils utilisés dans les applications JEE?
- (Q2) : Quel est l'impact de l'application de ces patrons JEE sur la modifiabilité des applications ?

Les données collectées pour répondre à la question Q1 ont été analysées et discutées dans la section 4.1. En résumé, même s'il y a très peu de littérature sur les patrons JEE, nous avons constaté que toutes les applications JEE étudiées appliquent un certain nombre de ces patrons. De plus, les patrons JEE les plus utilisés dans ces applications sont : 1) les patrons qui reflètent de bonnes pratiques de l'architecture en couches et qui ne sont pas spécifiques au *Framework* JEE, par exemple : les patrons « *Façade* », « *Application Service* », « *Business Object* » et « *DAO* » ou 2) les patrons qui sont renforcés par la technologie JEE, par exemple : l'application du « *Front Controller* » passe par l'utilisation des *servlets*. Les patrons les moins utilisés sont : 1) ceux qui ne constituent pas une pratique courante dans le développement d'applications en couches, exemple : les patrons « *Business Delegate* » et « *Application Controller* » ou 2) ceux qui dépendent de l'utilisation du modèle de composants EJB, exemple : le patron « *Composite Entity* ».

Les données collectées pour répondre à la question Q2 ont été analysées et discutées dans les sections 4.2 et 4.3. En résumé, notre analyse des données a abouti à très peu de corrélations entre l'existence des occurrences des patrons étudiés et la modifiabilité des applications analysées. Même, dans le cas de certains patrons visant à diminuer le couplage entre les couches de l'application (exemple : les patrons « *Façade* » et « *DAO* »), nous n'avons pas obtenu des corrélations significatives avec les métriques de couplage. Ces résultats peuvent s'expliquer par les facteurs suivants :

- Nous évaluons l'impact des patrons sur la qualité au niveau système et cela peut biaiser les résultats. En fait, chaque application inclut plusieurs occurrences de différents patrons JEE et donc les résultats trouvés ne reflètent pas l'impact d'un seul patron, mais plutôt l'impact de tous les patrons JEE qui ont été utilisés.
- Nous évaluons la qualité des applications analysées en fonction du nombre de patrons utilisés dans ces applications. Notre analyse ne tient pas compte du fait que ces patrons aient été correctement utilisés ou non.
- Dans le cas de l'analyse de plusieurs versions de la même application, nous évaluons l'évolution des métriques de modifiabilité en fonction des occurrences introduites ou supprimées des patrons JEE. Cependant, en passant d'une version à une autre, les développeurs ajoutent aussi de nouvelles fonctionnalités et de nouveaux composants. Ceci influence les métriques de modifiabilité et ne nous permet pas d'isoler l'impact des patrons JEE introduits dans la nouvelle version.

En nous basant sur notre étude approfondie des patrons JEE et notre expérience résultant de l'analyse manuelle de plusieurs applications JEE, nous déduisons que l'utilisation des patrons JEE facilite la compréhension, la maintenabilité et par conséquent la modifiabilité des applications. Notre déduction se base sur l'ensemble des faits constatés lors de notre étude :

- La majorité des patrons JEE sont implémentés par une seule classe telle que « *Business Delegate* », « *Application Service* », ce qui facilite leur utilisation et leur compréhension.
- Plusieurs patrons, tels que « *Session Facade* » et « *Data Access Object* », permettent une séparation des différentes logiques de l'application (présentation, métier et persistance).

- Selon son rôle, chaque patron JEE appartient à une couche spécifique de l'application. Ceci facilite la délimitation des couches de l'architecture d'un système et donc la compréhension et l'analyse de cette architecture.

5.2 Limites de l'étude

Plusieurs facteurs ont un impact sur la validité des résultats de notre étude. Ces facteurs sont discutés dans cette section.

5.2.1 Validité externe des résultats de l'étude

La validité externe d'une étude fait référence à la généralisation des résultats indépendamment des conditions ou paramètres de l'étude. La validité externe des résultats de notre étude est limitée par deux facteurs :

- L'échantillon des applications JEE analysées peut ne pas être représentatif des applications JEE en général. À cause de la non-disponibilité du code source d'applications JEE, nous n'avons analysé que 17 applications couvrant différents domaines et ayant différentes tailles. Cependant, il s'agit à ce jour de la plus grande étude sur ce sujet.
- L'échantillon des patrons JEE étudiés inclut les patrons supportant la modifiabilité. Ce choix se justifie par le fait que notre étude se concentre sur la modifiabilité des applications. Il serait intéressant d'étudier d'autres patrons JEE et évaluer leur impact sur la modifiabilité.

5.2.2 Validité interne de l'étude

La validité interne de l'étude fait référence à la capacité de l'étude à produire des résultats attribuables à la variable indépendante définie dans l'étude et non pas à d'autres facteurs ou erreurs systématiques découlant du déroulement de l'étude. Dans notre contexte, nous identifions deux facteurs qui limitent la validité interne de notre étude :

- La validation des occurrences détectées des patrons JEE est basée sur une analyse manuelle du code des applications. Dans certains cas, nous avons interprété l'intention des développeurs pour décider si le patron est appliqué ou non. Ce problème est du au fait que plusieurs patrons JEE sont implémentés par une seule classe et que le développeur ne fait pas de référence explicite ni dans les commentaires du code, ni à travers le nom de la classe, au patron appliqué. Cela va influencer nos résultats et plus précisément le nombre de fausses occurrences trouvées ainsi que le nombre d'occurrences positives non trouvées. Cependant, nous n'avons considéré les occurrences comme vraies positives que lorsque les preuves étaient irréfutables.
- Pour chaque métrique, un seul outil est utilisé pour calculer les valeurs de cette métrique. Malgré que ces métriques ont des définitions formelles dans la littérature, les outils peuvent avoir différentes implémentations et interprétations de chaque métrique. Cela influence les résultats de notre étude. Nous n'avons pas comparé les valeurs retournées par les différents outils. Ceci ne fait pas partie de l'étendue de notre étude mais il peut être exploré davantage dans les travaux futurs.

CONCLUSION & TRAVAUX FUTURS

Rappelons que l'objectif général de notre travail est d'étudier l'utilisation des patrons de conception et d'évaluer l'impact de ces patrons sur la qualité des applications. En particulier, l'étendue de notre travail porte sur l'étude des applications ayant une architecture logicielle en couche (plus précisément les applications JEE) et à l'évaluation de l'impact de l'utilisation des patrons sur un seul attribut de qualité (la modifiabilité). Pour ce faire, nous avons identifié les patrons JEE qui supportent la modifiabilité et par la suite nous avons effectué une étude empirique dont le but est d'identifier les patrons JEE utilisés dans les applications JEE analysées et d'évaluer leurs impacts sur la modifiabilité des applications JEE.

Vu qu'il n'existe pas d'outils qui visent spécifiquement la détection des patrons JEE, nous avons procédé de cette façon :

i. Nous avons utilisé *Ptidej (Guéhéneuc et al., 2005)* pour décrire et détecter un sous-ensemble de patrons JEE (exemple : le patron « *Façade* »).
ii. Nous avons développé un module d'analyse « *Parseur* » qui parcourt un arbre syntaxique représentant le code source pour y identifier des nœuds dont les noms contiennent des mots particuliers, ou des nœuds qui implémentent certaines interfaces du *Framework* JEE.
iii. Nous avons utilisé un outil de rétro-ingénierie pour produire des modèles à partir du code source des applications analysées. Nous avons analysé ces modèles pour valider les occurrences de patrons retournées par Ptidej et par notre parseur.
iv. Nous avons fait une analyse manuelle du code source pour identifier les occurrences positives non trouvées.

Nous constatons que même s'il y a très peu de littérature sur les patrons JEE, ces derniers ont été appliqués dans les applications JEE analysées, mais avec un degré d'utilisation qui varie d'une application à l'autre. En ce qui concerne les corrélations entre l'existence des occurrences des patrons étudiés et la modifiabilité des applications analysées, notre analyse a

abouti à très peu de corrélations qui n'incluent pas les corrélations attendues. Cependant, notre étude approfondie des patrons JEE et notre expérience résultant de l'analyse de plusieurs applications JEE nous mènent à conclure que l'utilisation des patrons JEE permet la compréhension, la maintenabilité et par conséquent la modifiabilité des applications.

En termes de perspectives à court terme, ce travail peut être vu comme étant une étude préliminaire qui peut être étendue : (i) en augmentant le nombre d'applications JEE à analyser, (ii) en augmentant le nombre de patrons à étudier, et (iii) en étudiant l'impact des patrons JEE sur la qualité des systèmes, mais au niveau local afin de minimiser l'impact cumulatif résultant de la somme de l'impact de tous les patrons appliqués.

À moyen terme, pour pallier au problème de disponibilité des applications JEE, nous prévoyons mener une étude avec un groupe d'étudiants qui développent des applications JEE. Cette étude aurait pour but d'évaluer la qualité de ces applications avant et après utilisation de patrons JEE.

À long terme, nous pouvons : (i) étendre l'outil *Ptidej* (Guéhéneuc et al., 2005) pour qu'il soit capable de détecter les patrons JEE ou développer notre propre outil dédié à la détection des patrons JEE et (ii) proposer une nouvelle approche pour la restructuration des architectures en couches en utilisant les patrons JEE.

ANNEXE I

CORRESPONDANCES ENTRE LES PATRONS JEE ET LES TACTIQUES DE MODIFIABILITÉ

Dans cette section, nous présentons en détail les descriptions de patrons JEE et celle de tactiques de modifiabilité qui nous ont aidés pour déterminer l'existence de correspondance entre eux.

Tableau-A I-1 Détails de correspondance entre le patron « *Intercepting Filter* » et les tactiques de modifiabilité

Patron JEE : « Intercepting Filter »		
Description courte : Effectuer les prétraitements et les post-traitements des requêtes client.		
Tactiques supportées	Réduire la taille d'un module	Créer des composants faiblement couplés (les filtres) dont chacun d'eux est responsable d'un traitement spécifique.
	Limiter les dépendances	Limiter l'accès aux filtres en précisant la liste de filtres à exécuter pour chaque traitement.
	Restructurer	Créer des filtres indépendants dont chacun spécifique à un traitement particulier, ce qui diminue l'existence du code dupliqué

Tableau-A I-2 Détails de correspondance entre le patron « *Front Controller* » et les tactiques de modifiabilité

Patron JEE : « *Front Controller* »		
Description courte : Spécifier un point d'accès centralisé pour la manipulation des demandes.		
Tactiques supportées	Restructurer	éviter la duplication de la logique de contrôle.
	Abstraire les services communs	Regrouper les traitements de contrôle communs.

Tableau-A I-3 Détails de correspondance entre le patron « *Context Object* » et les tactiques de modifiabilité

Patron JEE : « *Context Object* »		
Description courte : Éviter l'utilisation des informations relatives au protocole hors de son contexte approprié.		
Tactique supportée	Encapsuler	Encapsuler les informations relatives à chaque protocole dans un composant indépendant

Tableau-A I-4 Détails de correspondance entre le patron « *Application Controller* » et les tactiques de modifiabilité

Patron JEE : « Application Controller »		
Description courte : Centraliser et modulariser la gestion des actions et des vues.		
Tactiques supportées	Réduire la taille d'un module	Améliorer la modularité du système.
	Augmenter la cohésion	
	Encapsuler	Encapsuler la gestion des actions et des vues dans un même contrôleur.

Tableau-A I-5 Détails de correspondance entre le patron « *View Helper* » et les tactiques de modifiabilité

Patron JEE : « *View Helper* »		
Description courte : Séparer la logique métier des vues et contenir le format des données qui va être présenté à l'utilisateur.		
Tactiques supportées	Réduire la taille d'un module	Séparer la logique métier des fonctionnalités d'affichage.
	Augmenter la cohésion	
	Encapsuler	Encapsuler la logique métier dans des composants '*helpers*' et les traitements relatifs à l'affichage dans des composants '*views*'.
	Utiliser un intermédiaire	Éviter l'incorporation directe des sous-vues dans les vues du système en créant des vues composites

Tableau-A I-6 Détails de correspondance entre le patron « *Composite View* » et les tactiques de modifiabilité

Patron JEE : « *Composite View*»		
Description courte : Créer des sous-vues atomiques et les combinés pour former des vues composites et gérer leurs contenus et leurs traitements de façon indépendante.		
Tactique supportée	Restructurer	Créer des vues composites composées de plusieurs sous-vues atomiques réduites l'existence du code dupliqué.

Tableau-A I-7 Détails de correspondance entre le patron « *Service to Worker* » et les tactiques de modifiabilité

Patron JEE : « *Service to Worker* »	
Description courte : Manipuler, contrôler et exécuter la commande avant de l'envoyer à la vue (c.-à-d.. l'afficher à l'utilisateur).	
Tactique supportée	Ce patron est une combinaison d'autres patrons JEE qui sont « *Front Controller* », « *Application Controller* » et « *View Helper* ». Vu que ces derniers supportent la modifiabilité, le patron « *Service to Worker* » supporte aussi la modifiabilité

Tableau-A I-8 Détails de correspondance entre le patron « *Dispatcher View* » et les tactiques de modifiabilité

Patron JEE : « *Dispatcher View* »		
Description courte : Traiter la demande, générer la réponse (en effectuant un ensemble de traitements limité) et gérer les vues.		
Tactique supportée	Augmenter la cohésion	Limiter les traitements effectués sur la demande et se concentre à la génération des réponses (l'affichage).

Tableau-A I-9 Détails de correspondance entre le patron « *Business Delegate* » et les tactiques de modifiabilité

Patron JEE : « *Business Delegate* »		
Description courte : Réduire le couplage entre la couche présentation et la couche métier en cachant les détails d'implémentations des services.		
Tactiques supportées	Utiliser un intermédiaire	fournir un point d'accès unique aux services de la couche métier.
	Restructurer	centraliser les changements.

Tableau-A I-10 Détails de correspondance entre le patron « *Service Locator* » et les tactiques de modifiabilité

Patron JEE : « *Service Locator* »		
Description courte : Localiser les composants ainsi que les services métiers de façon uniforme.		
Tactiques supportées	Encapsuler	Encapsuler la recherche des services, la complexité des traitements de consultation et la création de l'objet dans un unique composant.
	Restructurer	Centraliser les traitements de localisation des services (éviter la duplication de code)

Tableau-A I-11 Détails de correspondance entre le patron « *Session Facade* » et les tactiques de modifiabilité

Patron JEE : « *Session Facade* »		
Description courte : Regrouper et exposer les composants et les services métier en contrôlant l'accès à ces éléments.		
Tactiques supportées	Encapsuler	Encapsuler toute la logique métier ainsi que la complexité des interactions entre les objets métier dans un seul composant.
	Limiter les dépendances	Empêcher l'accès direct des clients aux composants métier.
	Utiliser un intermédiaire	Introduire un intermédiaire entre les clients et les composants métier.
	Restructurer	Réduire la complexité du code client.

Tableau-A I-12 Détails de correspondance entre le patron « *Application Service* » et les tactiques de modifiabilité

Patron JEE : « Application Service »		
Description courte : Centraliser la logique métier commune dans une couche de service.		
Tactiques supportées	Encapsuler	Créer une couche de services en encapsulant la logique métier commune à tous les « *Business Objects* ».
	Restructurer	Éliminer le code dupliqué en regroupant la logique métier commune.

Tableau-A I-13 Détails de correspondance entre le patron « *Business Object* » et les tactiques de modifiabilité

Patron JEE : « *Business Object* »		
Description courte : Séparer la logique métier de la logique de persistance.		
Tactiques supportées	Réduire la taille d'un module	Séparer la logique métier de la logique de persistance.
	Augmenter la cohésion	

Tableau-A I-14 Détails de correspondance entre le patron « *Composite Entity* » et les tactiques de modifiabilité

Patron JEE : « *Composite Entity* »		
Description courte : Représenter le modèle du domaine conceptuel comme étant une hiérarchie d'objets persistants.		
Tactiques supportées	Limiter les dépendances	Éliminer les dépendances entre les beans en centralisant la gestion des relations des différentes beans entités dans des beans composites.
	Restructurer	Réduire la complexité du code en diminuant le nombre de beans entités fines et en créant des beans entités composites.

Tableau-A I-15 Détails de correspondance entre le patron « *Transfert Object* » et les tactiques de modifiabilité

Patron JEE : « *Transfer Object* »		
Description courte : Transférer des données entre la couche présentation et la couche métier.		
Tactiques supportées	Restructurer	Minimiser l'existence du code dupliqué et réduire la complexité du code source en créant des composants qui sont responsable à extraire et à envoyer les informations des objets.

Tableau-A I-16 Détails de correspondance entre le patron « *Transfer Object Assembler*» et les tactiques de modifiabilité

Patron JEE : « Transfert Object Assembler»		
Description courte : Construire un modèle d'application qui regroupe plusieurs « Transfer Object »		
Tactiques supportées	Limiter les dépendances	Minimiser l'existence du code dupliqué et réduire la complexité du code source en créant des composants qui sont responsable à extraire et à envoyer les informations des objets.

Tableau-A I-17 Détails de correspondance entre le patron « *Value List Handler* » et les tactiques de modifiabilité

Patron JEE : « Value List Handler »		
Description courte : Assurer la recherche, la mise en cache, la sélection et l'examen des éléments d'une liste.		
Tactiques supportées	Augmenter la cohésion	Maintenir la logique métier dans les composants de la couche métier et l'accès aux données dans le « DAO ».
	Encapsuler	Encapsuler le comportement de gestion de la liste dans la couche métier.

Tableau-A I-18 Détails de correspondance entre le patron « *Data Access Object* » et les tactiques de modifiabilité

Patron JEE : « *Data Access Object* »		
Description courte : Réduire le couplage entre la couche métier et la couche persistance en encapsulant la partie relative à l'accès aux données.		
Tactiques supportées	Augmenter la cohésion	Encapsuler les fonctionnalités relatives à l'accès à la source de données et les manipuler dans une couche séparée
	Utiliser un intermédiaire	Agir comme un intermédiaire entre l'application et la source de données
	Restructurer	Réduire la complexité du code source des composants qui ont besoin d'accéder à la source de données

Tableau-A I-19 Détails de correspondance entre le patron « *Domain Store* » et les tactiques de modifiabilité

Patron JEE : « *Domain Store* »		
Description courte : Séparer la logique de persistance des objets métier.		
Tactiques supportées	Réduire la taille d'un module	Séparer la logique métier de la logique de persistance.
	Augmenter la cohésion	

Tableau-A I-20 Détails de correspondance entre le patron « *Web Service Broker* » et les tactiques de modifiabilité

Patron JEE : « Web Service Broker »		
Description courte : Fournir l'accès aux différents services de l'application en utilisant les protocoles XML et Web.		
Tactiques supportées	Utiliser un intermédiaire	Introduire une couche entre le client et les services externes.

ANNEXE II

VALIDATION DE LA NON-NORMALITE DES MÉTRIQUES POUR LE CHOIX DE COÉFFICIENT DE CORRÉLATION

Dans cette section, nous présentons les histogrammes résultant de la validation de la normalité des métriques. Nous les regroupons en catégorie de métriques.

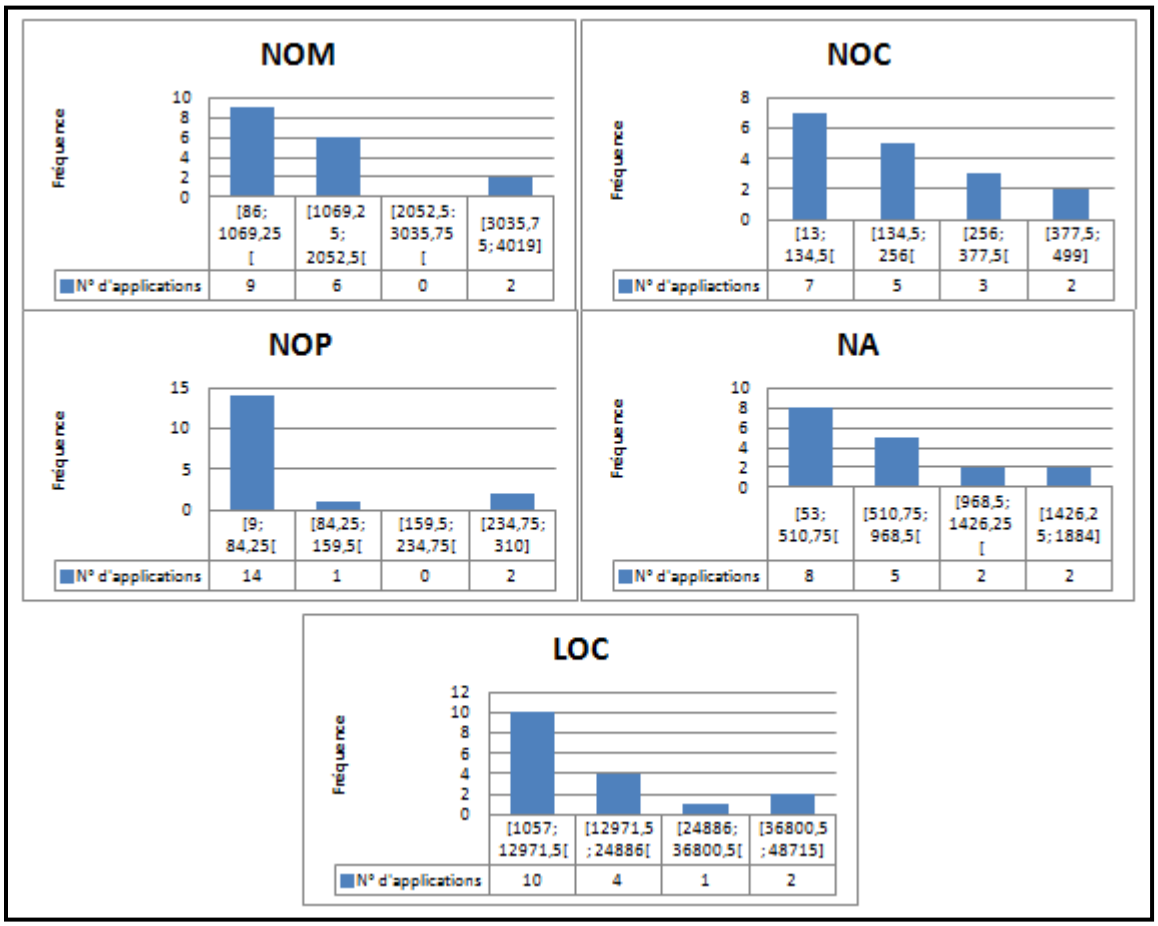

Figure-A II-1 Histogrammes de fréquence des métriques de taille (NOM, NOC, NOP, NA et LOC)

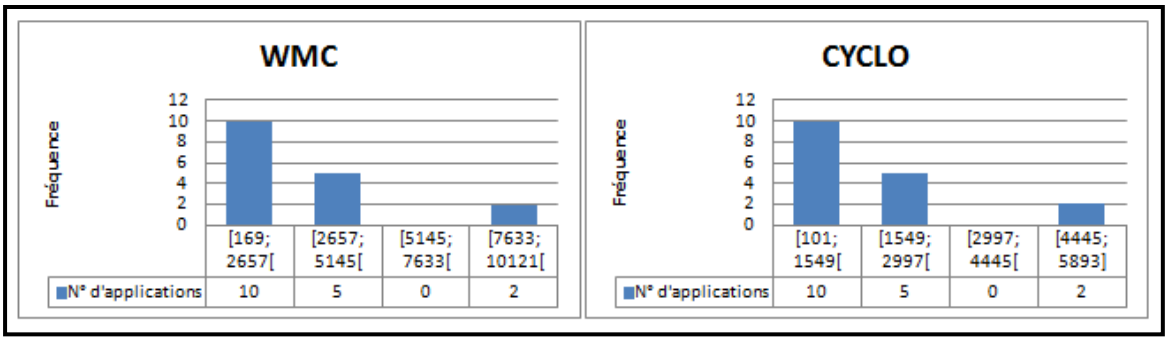

Figure-A II-2 Histogrammes de fréquence des métriques de complexité (WMC et CYCLO)

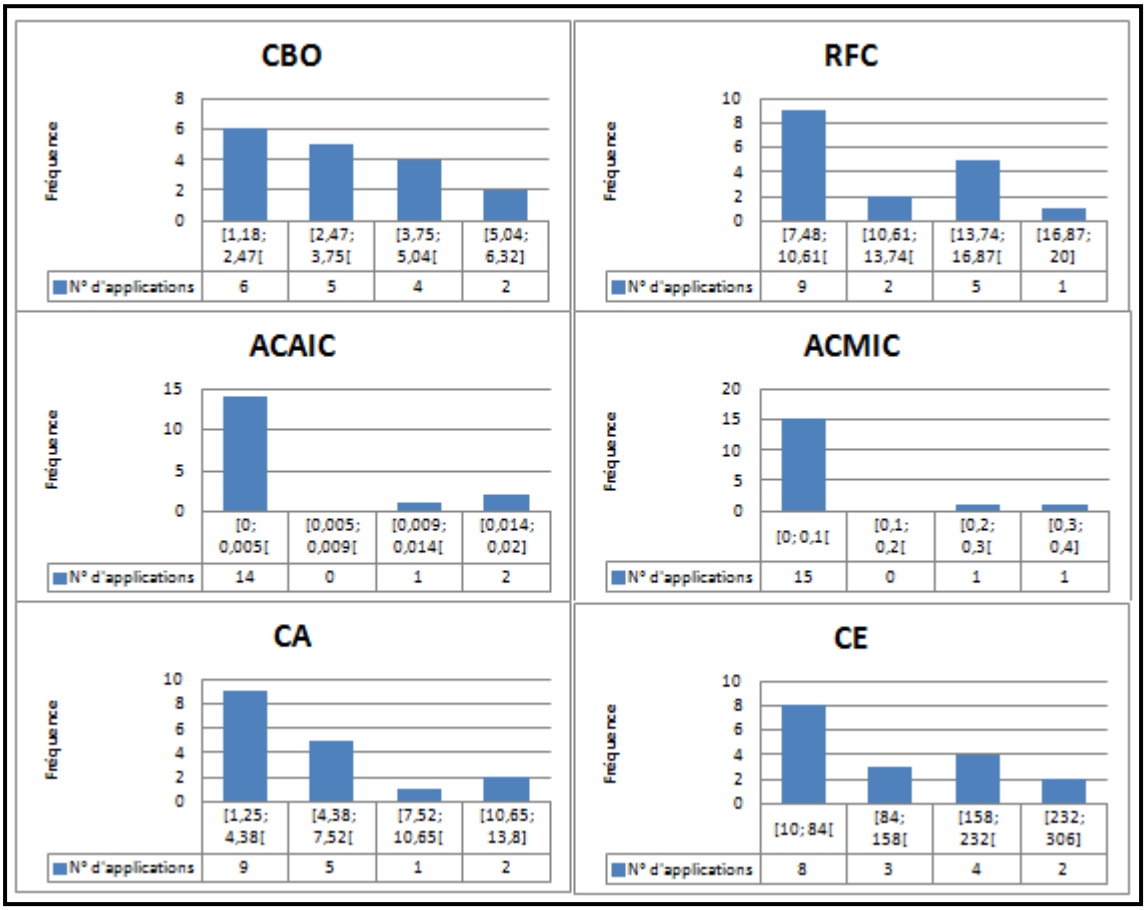

Figure-A II-3 Histogrammes de fréquence des métriques de couplage (VBO, RFC, ACAIC, CA et CE)

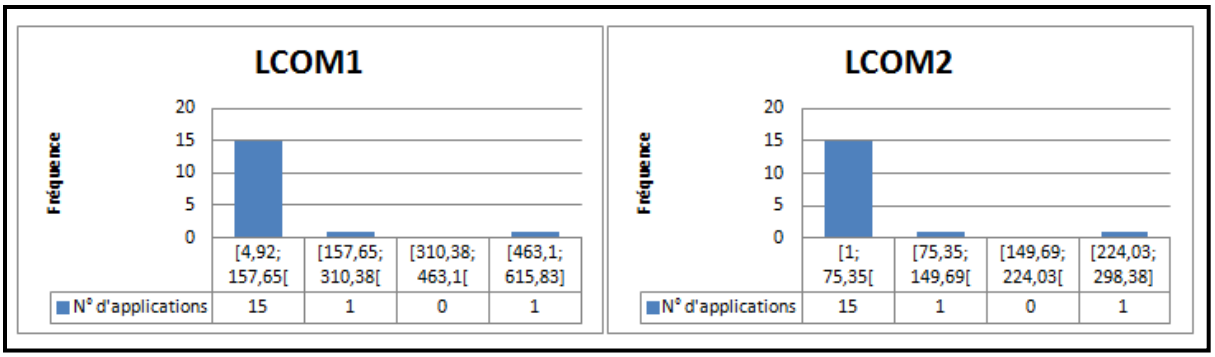

Figure-A II-4 Histogrammes de fréquence des métriques de cohésion (LCOM1, LCOM2)

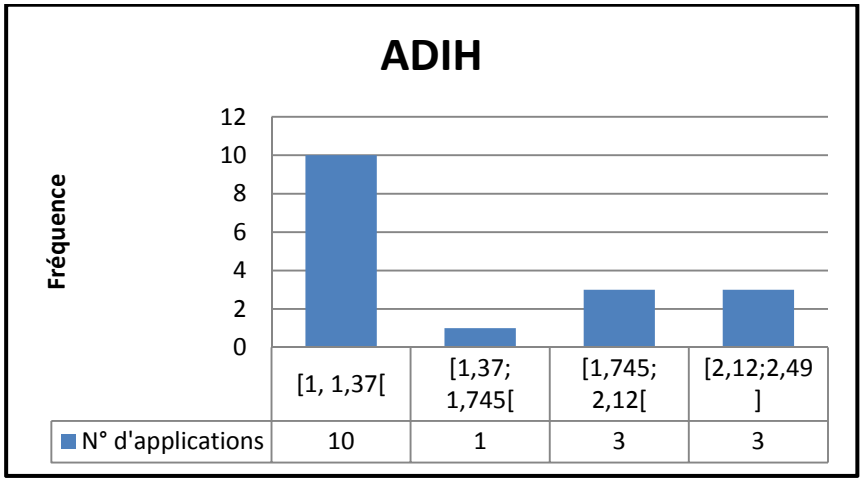

Figure-A II-4 Histogramme de fréquence de la métrique d'héritage ADIH

LISTE DE RÉFÉRENCES BIBLIOGRAPHIQUES

Abran, A., Bourque, P., Dupuis, R., & Moore, J. W. 2004. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*.

Ali, Mawal, et Mahmoud O Elish. 2013. « A Comparative Literature Survey of Design Patterns Impact on Software Quality ». In *2013 International Conference on, Information Science and Applications (ICISA)*. p. 1-7. IEEE.

Alur, Deepak, Dan Malks, John Crupi, Grady Booch et Martin Fowler. 2003. 2nd édition. *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*. Sun Microsystems, Inc.

Bachmann, Felix, Len Bass et Robert Nord. 2007. *Modifiability tactics*. DTIC Document.

Bass, Len, Paul Clements et Rick Kazman. 2003. 2nd édition. *Software architecture in practice*. Addison-Wesley Professional.

Bass, Len, Paul Clements et Rick Kazman. 2012. 3rd édition. *Software architecture in practice*. Addison-Wesley.

Bieman, James M, et Byung-Kyoo Kang. 1995. « Cohesion and reuse in an object-oriented system ». In *ACM SIGSOFT Software Engineering Notes*. Vol. 20, p. 259-262. ACM.

Bieman, James M, Greg Straw, Huxia Wang, P Willard Munger et Roger T Alexander. 2003. « Design patterns and change proneness: An examination of five evolving systems ». In *Proceedings. Ninth international. Software metrics symposium, 2003.* p. 40-49. IEEE.

Bosquet, Laurent. « Méthodologie de la recherche ». < http://staps.univ-lille2.fr/fileadmin/user_upload/ressources_peda/prepa_kine/metho_recherche.pdf >. Consulté le 21 juin 2015.

Briand, Lionel C, John W Daly et Jürgen Wüst. 1998. « A unified framework for cohesion measurement in object-oriented systems ». *Empirical Software Engineering,* vol. 3, no 1, p. 65-117.

Briand, Lionel C, Jürgen Wüst, John W Daly et D Victor Porter. 2000. « Exploring the relationships between design measures and software quality in object-oriented systems ». *Journal of systems and software,* vol. 51, no 3, p. 245-273.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad et Michael Stal. 1996. *Pattern-oriented software architecture, volume 1: A system of patterns*. John Wiley and Sons.

Chang, Chih-Hung, Chih-Wei Lu, William C Chu, Nien-Lin Hsueh et Chorng-Shiuh Koong. 2009. « A case study of pattern-based software framework to improve the quality of software development ». In *Proceedings of the 2009 ACM symposium on Applied Computing*. p. 443-447. ACM.

Chidamber, Shyam R, et Chris F Kemerer. 1994. « A metrics suite for object oriented design ». *IEEE Transactions on, Software Engineering,* vol. 20, n° 6, p. 476-493.

Clements, Paul, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord et Judith Stafford. 2010. *Documenting software architectures: views and beyond*. Pearson Education.

Correia, José Pedro, Yiannis Kanellopoulos et Joost Visser. 2009. « A survey-based study of the mapping of system properties to iso/iec 9126 maintainability characteristics ». In *IEEE International Conference on. Software Maintenance, 2009. ICSM 2009.* p. 61-70. IEEE.

Dagpinar, Melis, et Jens H Jahnke. 2003. « Predicting maintainability with object-oriented metrics-an empirical comparison ». In *2013 20th Working Conference on Reverse Engineering (WCRE)*. p. 155-155. IEEE Computer Society.

Dean Leffingwell, Don Widrig (544). 2003. *Managing Software Requirements: A Use Case Approach* 2nd Edition.

Di Penta, Massimiliano, Luigi Cerulo, Yann-Gaël Guéhéneuc et Giuliano Antoniol. 2008. « An empirical study of the relationships between design pattern roles and class change proneness ». In *IEEE International Conference on. Software Maintenance, 2008. ICSM 2008.* p. 217-226. IEEE.

Fontana, Francesca Arcelli, Vincenzo Ferme, Alessandro Marino, Bartosz Walter et Pawel Martenka. 2013. « Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains ». In *29th IEEE International Conference on. Software Maintenance (ICSM), 2013*. p. 260-269. IEEE.

Gamma, Erich, Richard Helm, Ralph Johnson et John Vlissides. 1994. *Design patterns: elements of reusable object-oriented software*. Pearson Education.

Garlan, David, et Mary Shaw. 1993. « An introduction to software architecture ». *Advances in software engineering and knowledge engineering,* vol. 1, p. 1-40.

Gautheir, Thomas D. 2001. « Detecting trends using Spearman's rank correlation coefficient ». *Environmental Forensics,* vol. 2, n° 4, p. 359-362.

Gilart-Iglesias, Virgilio, Francisco Maciá Pérez, Antonio Hernández Sáez, Diego Marcos-Jorquera et Juan Manuel García Chamizo. 2005. « A model for developing j2ee applications based on design patterns ». In *IADIS AC*. p. 352-360.

Grand, Mark. 2002. *Patterns in Java: A Catalogue of Reusable Design Patterns Illustrated with UML*. Wiley.

Guéhéneuc, Yann-Gaël, Jean-Yves Guyomarc'h, Duc-Loc Huynh, Olivier Kaczor, Naouel Moha, Samah Rached et Ptidej Team. 2005. « Ptidej: A Tool Suite ».

Hammouda, Imed, et Kai Koskimies. 2002. « A pattern-based J2EE application development environment ». *Nord. J. Comput.,* vol. 9, n° 1, p. 248-260.

Hollander, Myles, Douglas A Wolfe et Eric Chicken. 2013. *Nonparametric statistical methods*. John Wiley & Sons.

Horstmann, Cay. 2009. *Object-oriented design and patterns*. John Wiley & Sons.

Jean-Michel, Doudoux. 1999. « Développons en Java ». < http://www.jmdoudoux.fr/java/dej/chap-j2ee-javaee.htm >. Consulté le 21juin 2015.

Jeanmart, Sebastien, Yann-Gael Gueheneuc, Houari Sahraoui et Naji Habra. 2009. « Impact of the visitor pattern on program comprehension and maintenance ». In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. p. 69-78. IEEE Computer Society.

Johnson, Rod. 2004. *Expert one-on-one J2EE design and development*. John Wiley & Sons.

Kakarontzas, George, Eleni Constantinou, Apostolos Ampatzoglou et Ioannis Stamelos. 2013. « Layer assessment of object-oriented software: A metric facilitating white-box reuse ». *Journal of Systems and Software,* vol. 86, n° 2, p. 349-366.

Kazman, Rick, Len Bass et Mark Klein. 2006. « The essential components of software architecture design and analysis ». *Journal of Systems and Software,* vol. 79, n° 8, p. 1207-1216.

Khomh, Foutse, et Y-G Guéhéneuc. 2008. « Do design patterns impact software quality positively? ». In *12th European Conference on. Software Maintenance and Reengineering, 2008. CSMR 2008.* p. 274-278. IEEE.

Khomh, Foutse, Yann-Gaël Guéhéneuc et Giuliano Antoniol. 2009. « Playing roles in design patterns: An empirical descriptive and analytic study ». In *IEEE International Conference on*. *Software Maintenance, 2009. ICSM 2009.* p. 83-92. IEEE.

Larman, Craig, Emmanuelle Burr, Marie-Cécile Baland et Luc Carité. 2005. *UML 2 et les Design Patterns*. Pearson Education.

Losavio, Francisca, Ledis Chirinos, Nicole Lévy et Amar Ramdane-Cherif. 2003. « Quality characteristics for software architecture ». *Journal of Object Technology,* vol. 2, n° 2, p. 133-150.

Mouratidou, Maria, Vassilios Lourdas, Alexander Chatzigeorgiou et Christos K Georgiadis. 2010. « An assessment of design patterns' influence on a Java-based e-commerce application ». *Journal of theoretical and applied electronic commerce research,* vol. 5, n° 1, p. 25-38.

Ng, TH, SC Cheung, WK Chan et YT Yu. 2006. « Toward effective deployment of design patterns for software extension: a case study ». In *Proceedings of the 2006 international workshop on Software quality*. p. 51-56. ACM.

Niziaek, A, Wojciech Zabierowski et Andrzej Napieralski. 2008. « Application of JEE 5 technologies for a system to support dental clinic management ». In *Proceedings of International Conference on*. *Modern Problems of Radio Engineering, Telecommunications and Computer Science, 2008.* p. 565-568. IEEE.

Pathak, R. P. 2011. *Statistics in Education and Psychology* (April 23, 2011). Pearson India, 180 p.

Ragab, Sahar Reda, et Hany H Ammar. 2010. « Object oriented design metrics and tools a survey ». In *The 7th International Conference on*. *Informatics and Systems (INFOS), 2010.* p. 1-7. IEEE.

Rakotomalala, Ricco. 2008. *Tests de normalité*. Coll. « Université Lumière Lyon 2 ». < http://eric.univ-lyon2.fr/~ricco/cours/cours/Test_Normalite.pdf >. Consulté le 21 juin 2015.

Roden, Patricia L, Shamsnaz Virani, Letha H Etzkorn et Sherri Messimer. 2007. « An empirical study of the relationship of stability metrics and the qmood quality models over software developed using highly iterative or agile software processes ». In *Seventh IEEE International Working Conference on*. *Source Code Analysis and Manipulation, 2007. SCAM 2007.* p. 171-179. IEEE.

Saraiva, Juliana. 2013. « A Roadmap for Software Maintainability Measurement ». *Proceedings of the 2013 International Conference on Software Engineering (Proceeding ICSE '13)*, p. 1453-1455.

Sheldon, Frederick T, Kshamta Jerath et Hong Chung. 2002. « Metrics for maintainability of class inheritance hierarchies ». *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 14, n° 3, p. 147-160.

Tahvildar, Ladan, et Kostas Kontogiannis. 2004. « Improving design quality using meta-pattern transformations: a metric-based approach ». *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 16, n° 4-5, p. 331-361.

Zhang, Cheng, et David Budgen. 2012. « What do we know about the effectiveness of software design patterns? ». *IEEE Transactions on, Software Engineering,* vol. 38, n° 5, p. 1213-1231.