Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Abbes Laghrour Khenchela
Faculté des Sciences et de la Technologie
Département des Mathématiques et d'informatique

Mémoire de fin d'études Pour l'obtention du diplôme de Master (L.M.D)

Spécialité : Informatique
Option : Génie Logiciel et Systèmes Distribués

Ce mémoire intitulé :

Une approche basée Patterns pour une transformation des modèles

- Réalisé par :

  Ghegaglia Athmane
  Aounallah Soheib

- Dérigé par :

  Dr. Messaoudi Nabil

2019/2020

remerciement

Nous tenons à remercier toutes les personnes qui ont contribué de près ou de loin à notre réussite dans notre cursus universitaire.

Nous voudrions dans un premier temps exprimer toute notre reconnaissance à Monsieur Messaoudi Nabil, Notre professeur encadreur. pour son professionnalisme, son dynamisme et surtout sa disponibilité pour toutes les sollicitations et aussi son sens d'écoute et de compréhension. Nous le remercions de nous avoir encadrés, orientés, aidés et conseillés.

À tous les professeurs, nous présentons nos remerciements, notre respect et notre gratitude.

Nous remercions également nos familles notamment nos très chers parents qui ont toujours été là pour nous soutenir, nous aider et nous encourager.

Résumé

Une bonne utilisation des modèles de conception conduit la construction de systèmes logiciels bien structurés, maintenables et réutilisable. Beaucoup de langages de transformation de modèles sont dénudés de la notion de design pattern. Nous avons utilisé un langage non dédié aux transformations de modèles pour prendre en charge une transformation de textes vers textes en basant sur une composition de Patterns.
Le langage utilisé est JAVA et les patterns utilisés sont Observer et Visitor.

Mots clé : Désign Pattern, Langage de Transformation de modèles, JAVA, Réutilisation.

Abstract

Good use of design models leads to the construction of well-structured, maintainable and reusable software systems, many model transformation languages are stripped of the concept of design pattern. We used a language not dedicated to model transformations to support text-to-text transformation.
The language used is JAVA and the patterns used are observer and visitor.

Keywords : Design Pattern, Modeling Transformation Language, JAVA, Reuse.

الملخص

يتم تجريد العديد من لغات تحويل النماذج من مفهوم نمط التصميم.
لقد استخدمنا لغة غير مخصصة لنمذجة التحولات لدعم تحويل النص إلى نص.
اللغة المستخدمة هي جافا والأنماط المستخدمة هي المراقب والزائر.

الكلمات المفتاحية : نمط المصممين، لغة تحويل النماذج، جافا، إعادة الإستخدام.

Sommaire

Table des figures

Introduction Générale

La réutilisation du code a fait l'objet de plusieurs articles de recherche, l'utilisation de Design Pattern en particulier les Patterns de Gang of Four pour remédié à ce problème.
L'absence de cette caractéristique au niveau des langages de transformation de modèles, nous a poussé à prendre cette caractéristique en utilisant les langages non dédiés aux transformation de modèle. Notre travail consiste à l'utilisation de deux Patterns du Gang of Four qui sont "Observer" et "Visitor" et d'incorporer dans un travail futur dans un IDE eclipse. La prise en charge de ce phénomène passera à travers l'organisation suivante de notre mémoire :

- Le chapitre 1 est consacré aux notions théorique des Design Patterns.
- Le chapitre 2 est consacré aux notions des transformations des modèles.

notre collaboration et une étude du cas sont définis respectivement dans les chapitres 3 et 4.

Chapitre 1

Les Patrons de Conception

1.1 Introduction

Les modèles de conception « décrivent un problème qui se pose sans cesse dans notre environnement. Décrivez ensuite le cœur de la solution à ce problème, de manière à ce que vous puissiez utiliser cette solution un million de fois, sans jamais faire la même chose deux fois [1]. » Dans l'ingénierie logicielle, les modèles de conception fournissent une expérience précieuse dans une syntaxe compréhensible par les ingénieurs logiciels afin de résoudre divers problèmes rencontrés pendant le développement. Une bonne utilisation des modèles de conception conduit à la construction de systèmes logiciels bien structurés, maintenables et réutilisables [2].

1.2 Les types des patterns

Il y a environ 26 Patterns actuellement découverts, ces 26 Patterns peuvent être classés en 3 types :

- Creational design patterns (Patterns créateurs)
- Structural design patterns (Patterns structuraux)
- Behavioral Design Patterns (Patterns comportementaux)

1.2.1 Creational design patterns

Les Patterns de conception créatifs (Creational design patterns) concernent la façon de créer des objets. Ces modèles de conception sont utilisés - lorsqu'une décision doit être prise au moment de l'instanciation d'une classe (c'est-à-dire la création d'un objet d'une classe). Dans cette classe, on a 6 Patterns :

- Factory Method Pattern
- Abstract Factory Pattern
- Singleton design pattern
- Prototype Design Pattern
- Builder Design Pattern
- Object Pool Pattern

1.2.2 Structural design patterns

Les modèles de conception structurelle (Structural design patterns) concernent la façon dont les classes et les objets peuvent être composés, pour former des structures plus grandes.

Ils simplifient la structure en identifiant les relations, ces modèles se concentrent sur la façon dont les classes héritent les unes des autres et comment elles sont composées à partir d'autres classes .
Dans cette classe on a 7 Patterns :

- Adapter Pattern
- Bridge Pattern
- Composite Pattern
- Decorator Pattern
- Facade Pattern
- Flyweight Pattern
- Proxy Pattern

1.2.3 Behavioral Design Patterns

Les modèles de conception comportementale (Behavioral Design Patterns) concernent l'interaction et la responsabilité des objets.

Dans ces modèles de conception, l'interaction entre les objets doit être telle qu'ils puissent facilement se parler et être toujours couplés de manière lâche.

Cela signifie que l'implémentation et le client doivent être couplés de manière lâche afin d'éviter le codage en dur et les dépendances . Dans cette classe on a 12 Patterns :

- Chain of Responsibility Pattern
- Command Pattern
- Interpreter Pattern
- Iterator Pattern
- Mediator Pattern
- Memento Pattern
- Observer Pattern

- State Pattern
- Strategy Pattern
- Template Pattern
- Visitor Pattern
- Null Object

Dans ce mémoire on va concentré sur deux Patterns qui font la base pour notre projet , les deux Patterns ce sont :

- Observer Pattern
- Visitor Pattern

1.3 Observer Pattern

Observer Pattern est un patron de conception de la famille des patrons comportementaux (Behavioral Design Patterns), il définit une relation entre les objets de type Un-à-Plusieurs, de façon que lorsqu'un objet change d'état, tous ce qui en dépendent en soient informés soient mis à jour automatiquement. . [20]
Ce Pattern met en place deux classes principales :

- la classe Observer : les instances de ces classes sont celles intéressées par les événements.
- la classe Source : les instances de ces classes sont celles générant des événements.

Implémentation

Le diagramme UML du pattern Observateur définit deux interfaces et deux classes.

L'interface Observateur sera implémenté par toutes classes qui souhaitent avoir le rôle d'observateur.

- C'est le cas de la classe ObservateurConcret qui implémente la méthode actualiser (Observable). Cette méthode sera appelée automatiquement lors d'un changement d'état de la classe observée.

- On trouve également une interface Observable qui devra être implémentée par les classes désireuses de posséder des observateurs.

- La classe ObservableConcret implémente cette interface, ce qui lui permet de tenir informer ses observateurs. Celle-ci possède en attribut un état (ou plusieurs) et un tableau d'observateurs. L'état est un attribut dont les observateurs désirent suivre l'évolution de ses valeurs. Le tableau d'observateurs correspond à la liste des observateurs qui sont à l'écoute. - En effet, il ne suffit pas à une classe d'implémenter l'interface Observateur pour être à l'écoute, il faut qu'elle s'abonne à un Observable via la méthode ajouterObservateur(Observateur).

- La classe ObservableConcret dispose de quatre méthodes :

- ajouterObservateur(Observateur)
- supprimerObservateur(Observateur)
- notifierObservateurs()
- getEtat()

Les deux premières permettent, respectivement, d'ajouter des observateurs à l'écoute de la classe et d'en supprimer.

- En effet, le pattern Observateur permet de lier dynamiquement (faire une liaison lors de l'exécution du programme par opposition à lier statiquement à la compilation) des observables à des observateurs.

- La méthode notifierObservateurs() est appelée lorsque l'état subit un changement de valeur. Celle-ci avertit tous les observateurs de cette mise à jour.

- La méthode getEtat() est un simple accesseur en lecture pour l'état. En effet, les observateurs récupèrent via la méthode actualiser(Observable) un pointeur vers l'objet observé. Puis, grâce à ce pointeur, et à la méthode getEtat() il est possible d'obtenir la valeur de l'état.

Le schéma suivant explique l'architecture de le pattern observer :

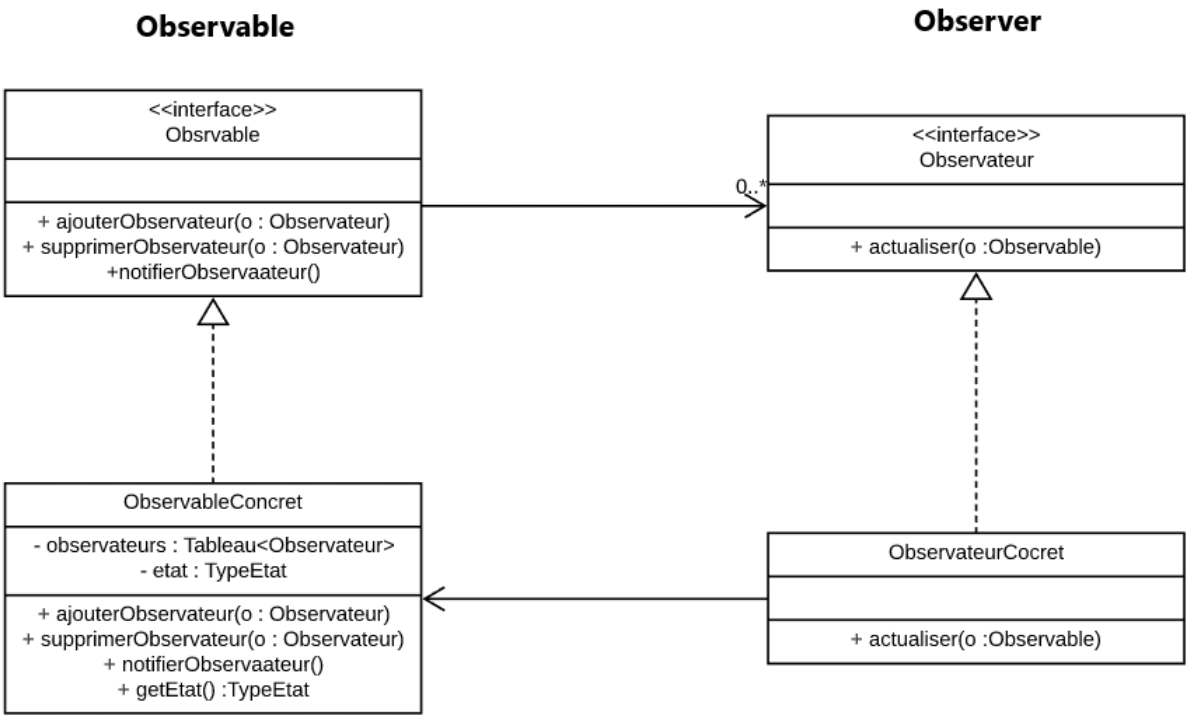

FIGURE 1.1 – UML pour le Pattern Observer

Avantages :

- Il décrit le couplage entre les objets et l'observateur.
- Il prend en charge la communication de type diffusion.

Usage :

- Lorsque le changement d'état dans un objet doit être reflété dans un autre objet sans maintenir les objets étroitement couplés.

1.4 Visitor Pattern

Le pattern Visitor permet de séparer de manière claire un algorithme d'une structure de donnée. L'algorithme n'est donc pas implémenté dans la classe mais dans des classes externes.
On limite ainsi tout couplage entre les données et leurs traitement ce qui permet d'ajouter des traitements a nos structures de données sans avoir a les modifier. De plus la classe externe est informé du type exact de l'objet qu'il visite a l'aide du principe du double dispatch . [20]

Implémentation :

Bien que souvent évoqué comme complexe ce design pattern est relativement simple. Notre classe visitable (c'est a dire notre structure de donnée) possède une méthode accept qui nous permet d'injecter notre visiteur (on précise une interface correspondant aux visiteurs voulu en type d'entrée).

En même temps les visiteurs concret implémente une interface qui contient une méthode visit par type d'objet visitable (ainsi on se sert du typages dynamique pour lors de l'utilisation appellera la bonne méthode – double dispatch).

Dans notre objet visitable on appellera donc la methode visit de notre visiteur en lui passant l'objet lui même.

Avec le pattern Visitor, on obtient deux hiérarchies distinctes :

- la hiérarchie des objets support de données
- la hiérarchie des Visiteurs (contenant les traitements sur les données)

Le schéma suivant explique l'architecture de le pattern Visitor :

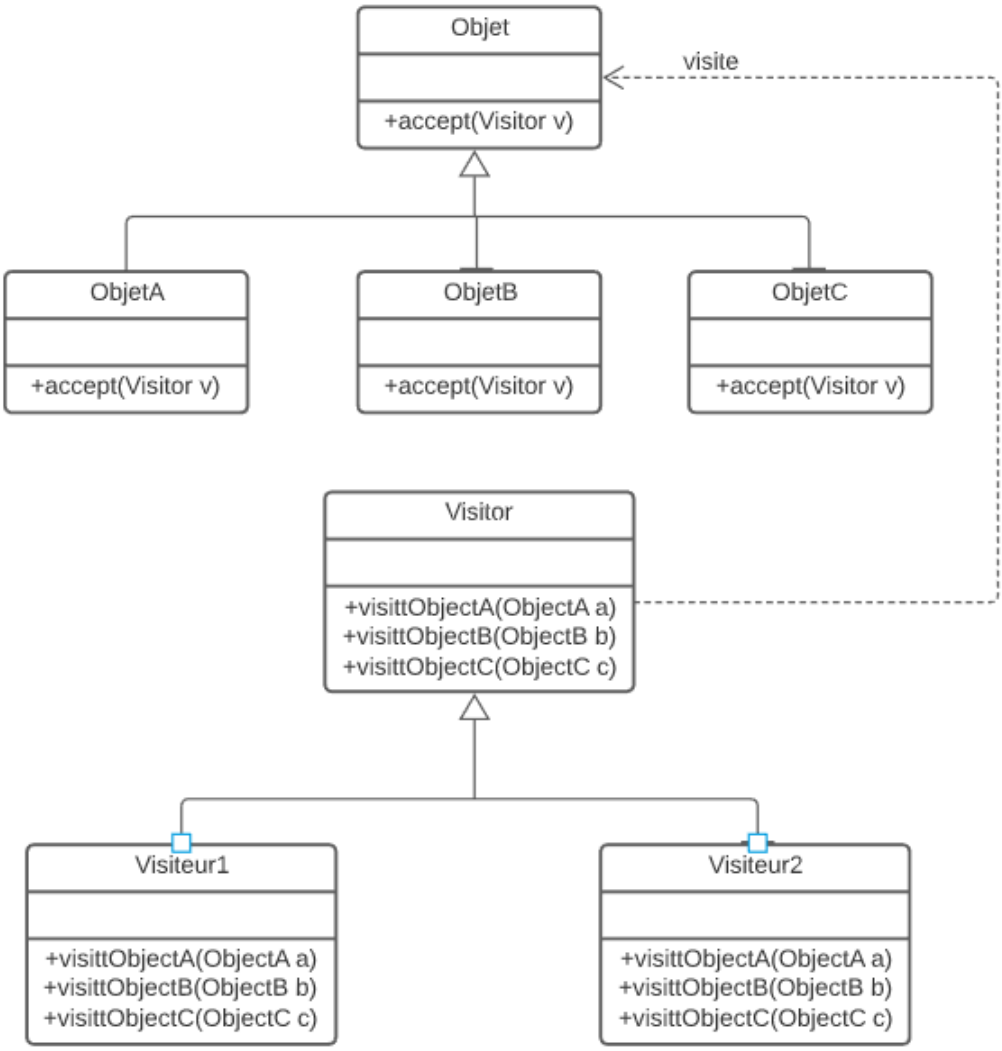

FIGURE 1.2 – UML pour le Pattern Visitor

D'un point de vue implémentation, cela nous donne : [21]

Pour appliquer un visiteur, on procède de la sorte :

Voici alors ce qu'il se passe (admettons que unObjet soit de type ObjetDeTypeA) :

1- la méthode accept de ObjetDeTypeA est appelée

2- la méthode visitObjetDeTypeA du visiteur v est alors appelée avec comme paramètre this (notre objet de type ObjetDeTypeA)

3- on entre alors dans le corps de la méthode visitObjetDeTypeA du visiteur. Le paramètre obj fait référence à l'objet de type ObjetDeTypeA

FIGURE 1.3 – Code d'un objet visité

FIGURE 1.4 – Implémentation d'un visiteur

sur lequel on applique notre traitement

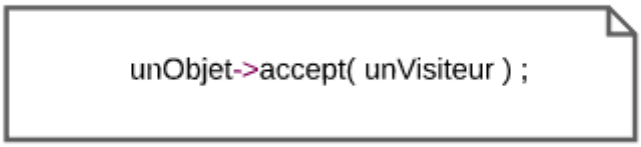

FIGURE 1.5 – Application d'un visiteur

Intérêt du pattern Visitor :

Pour ajouter un traitement à notre application, il suffit donc de créer un nouveau Visiteur avant de surcharger les méthodes permettant de visiter les différents objets de la hierarchie. Ensuite, pour utiliser le visiteur, on fait :

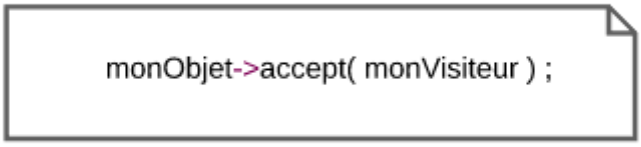

FIGURE 1.6 – Application d'un visiteur

De cette manière, on peut facilement ajouter de nouveaux traitements sans toucher à la hiérarchie de nos objets.

Avantages :

- le code est plus clair (des fonctionnalités différentes se trouvent dans des Visiteurs différents).
- des équipes différentes peuvent travailler sur des fonctionnalités différentes sans gêner les autres équipes.
- on n'est pas obligé de tout recompiler à chaque ajout d'une fonctionnalité (seul le code du Visiteur est recompilé).
- Ajout de nouvelles opérations très facile.

1.5 Modèles de conception orientés objet

L'étude la plus populaire sur les modèles de conception a été rapportée par Gamma et al. [3]. Dans cette étude, les auteurs ont proposé vingt trois modèles de conception standard orientés objet. En fait, avant Gamma et al., il y avait déjà des langages de programmation qui utilisaient des modèles de conception sans les nommer explicitement. Par exemple, la structure model-view-controller dans Smalltalk-80 est un exemple antérieur du modèle de conception de l'observateur [4].

Un modèle de conception doit « nommer, résumer et identifier les aspects clés d'une structure de conception commune qui le rendent utile pour créer un modèle réutilisable orienté objet ». Par conséquent, les modèles de conception qu'ils ont créés se concentrent spécifically sur la façon dont une solution précédemment identified peut être adoptée par une nouvelle implémentation utilisant un langage de programmation orienté objet. Un modèle de conception est essentiellement décrit par quatre éléments principaux :

1. son nom, qui résume succinctement l'objectif du modèle ;
2. le problème qu'il résout, qui décrit quand appliquer un modèle ;
3. la structure de la solution générale, qui énumère les participants du modèle : les éléments, leurs relations et les rôles qu'ils jouent ;
4. les conséquences, qui énumèrent les avantages, les inconvénients et les compromis à prendre en compte lors de l'utilisation du modèle

Une illustration d'un modèle de conception tiré du livre [3] est le modèle de conception « Stratégie ». Ce modèle de conception est appliqué lorsque de nombreuses classes connexes differ seulement selon certains comportements specific. Comme solution, le modèle de stratégie propose de déléguer la tâche des comportements specific à une interface, de les encapsuler dans des objets, et de laisser chaque comportement specific être mis en œuvre indépendamment dans l'isolement et de manière interchangeable. Les avantages sont :

1. les développeurs se retrouvent avec une famille de comportements connexes qui sont interchangeables ;
2. les développeurs éliminent les conditionnalités en créant une sous-classe different de l'interface

pour chaque comportement s'appuyant sur le polymorphisme. Cependant, les inconvénients sont :

1. l'augmentation du nombre d'objets créés ;
2. la notification du client, qui utilise l'interface, sur ces different behaviors.

Un modèle de conception doit être compréhensible afin d'être appliqué par tout développeur pour résoudre le même problème dans les contextes different. Par conséquent, Gamma et al. ont également utilisé les éléments suivants afin de représenter un modèle de conception plus précisément :

1. mieux comprendre le contexte : intention, motivation et applicabilité ;
2. faciliter la présentation de la demande : structure, participants et collaborations ;
3. pour montrer le modèle en action : mise en œuvre, code d'échantillon, utilisations connues et modèles connexes.

1.6 Modèles de conception dans d'autres paradigmes de programmation

Les modèles de conception sont également importants dans d'autres paradigmes de programmation. Diverses études ont publié des modèles de conception pour leurs paradigmes respectifs [5, 6, 7, 8, 9, 10]. Dans la plupart des études, le but de proposer des modèles de conception était :

1. documenter la solution d'un problème commun dans leur paradigme ;
2. proposer la recette pour résoudre le problème commun ;
3. automatiser une partie du développement du programme par génération de code. par Antoy et Hanus [5] ont présenté des modèles de conception pour les langages de logique fonctionnelle.

Ils ont présenté les modèles de conception five afin de résoudre certains problèmes généraux et difficiles dans les langages de logique fonctionnelle. Ils ont produit un catalogue en décrivant chaque modèle de conception avec ces caractéristiques : « nom », « intention », « applicabilité », « structure », « conséquences », « utilisations connues » et « voir aussi ». En outre, ils ont discuté en détail de chaque modèle de conception en fournissant un exemple d'implémentation dans le langage logique fonctionnel Curry.

Laemmel et Visser [9] ont considéré les modèles de conception comme des outils de communication de l'expertise en construction de programmes et ont proposé un ensemble de modèles de conception pour la programmation stratégique fonctionnelle. Ils ont produit un catalogue de 13 modèles de conception présentant les caractéristiques suivantes : « nom », « catégorie », « intention », « motivation », « applicabilité », « schéma », « description », « code d'échantillon », « conséquences » et « modèles connexes ». Ils ont utilisé le code Haskell pour représenter la structure de chaque modèle de conception. La programmation stratégique utilise des stratégies pour parcourir les divers termes du programme. Dans une autre étude de Laemmel et al. [8], ils ont fourni une abstraction de la stratégie de traversée. Bien que les stratégies de base ne soient pas mentionnées spéciquement Comme modèles de conception, ils sont des éléments primitifs qui résolvent de nombreux problèmes dans la programmation stratégique. Ils proposent quatre de ces éléments primitifs pour une stratégie de traversée.

Bagge et Laemmel [6] ont conçu un langage pour représenter comment traverser une structure arborescente/graphique. Ensuite, ils ont utilisé ce langage pour exprimer les stratégies de traversée indépendamment du langage de mise en œuvre. Le langage représente un pseudocode structuré qui peut être associé à un langage orienté objet. Ils ont décrit dix stratégies de traversée qui peuvent être utilisées comme primitives.

Visser [10] a mené une enquête sur les stratégies de réécriture de la transformation des programmes. Il a d'abord créé une taxonomie des techniques de transformation des programmes. Ensuite, il s'est concentré sur la façon dont les programmes sont représentés en créant les éléments de base pour spécifier les transformations de programme. Ces éléments de base sont utilisés comme éléments de base pour représenter diverses stratégies de réécriture de la transformation des programmes. Enfin, il a démontré comment les stratégies de réécriture de la transformation des programmes sont mises en œuvre à l'aide de ces éléments de base. En résumé, les études de modèles de conception existantes dans d'autres paradigmes de programmation se concentrent sur :

1. comment répartir les langues cibles en blocs de construction pour créer un langage générique représentant les modèles de conception ;
2. catalogage systématique des modèles de conception candidats format comme Gamma et al. [3] ;

3. afficher des exemples et des mises en œuvre des modèles de conception dans les langues cibles ;
4. utiliser les éléments constitutifs pour représenter des systèmes de conception plus complexes. Nous adoptons une approche similaire dans cette thèse tout en créant un catalogue de modèles de conception pour les transformations de modèles.

1.7 Limites des modèles de conception

Les modèles de conception sont utiles en termes de transfert d'une solution entre les implémentations different par un développeur. Toutefois, ils ont également certains échanges commerciaux et des restrictions [11]. La première conséquence de l'utilisation de modèles de conception est d'essayer d'appliquer beaucoup d'entre eux sur les mêmes objets. Cela nuit à l'entretien du projet. Il n'est pas clair combien de modèles de conception sont suffisants dans un projet. Il existe également un risque d'impact négatif sur le projet en appliquant trop de modèles de conception inutiles [12]. Cependant, il existe des preuves scientific pour prouver que les modèles de conception aident à la maintenance des projets lorsqu'ils sont utilisés de manière appropriée. Dans une étude de Prechelt et al. [13], les auteurs ont fait deux expériences qui ont prouvé que documenter l'existence de 21 modèles de conception dans le projet réduit considérablement le temps nécessaire pour effectuer l'entretien nécessaire du projet.

Les modèles de conception sont également appliqués pour éliminer les mauvaises pratiques de conception [14]. L'application de mauvaises pratiques de conception (p. ex., antipatterns) ou l'application de bonnes tendances de conception de mauvaise manière a un effet négatif sur les mesures de la qualité des logiciels [16]. Une mesure peut décourager l'intégration d'un modèle de conception dans le projet, ou vice versa. Par exemple, le modèle de « pont » diminue le niveau de profondeur dans une structure hiérarchique de classe [14]. Par conséquent, son application donne un meilleur design en termes de mesure de la qualité de profondeur d'héritage. Cependant, le modèle « Visiteur » permet au programme de traverser la structure de classe et de traiter chaque classe de manière efficient, tout en doublant le nombre de nouvelles classes dans le système. Les modèles de conception résolvent les problèmes de conception en en-

capsulant les abstractions via des objets, en se concentrant sur les interfaces plutôt que sur les implémentations, en améliorant les mécanismes de réutilisation et en déléguant les tâches aux objets les plus appropriés [3]. Cependant, ces avantages ne sont pas valables pour tous les modèles de conception. Selon l'étude de Khomh et Gueheneuce [15], le modèle « composite », qui code la hiérarchie en termes de composants et de composites, a un impact positif en termes d'extensibilité, de réutilisabilité, de compréhensibilité et d'attributs de qualité de l'évolutivité. Cependant, le modèle « Flyweight », qui réduit l'empreinte mémoire des gros objets, affiche un comportement négatif sur la plupart des attributs de qualité, à l'exception de l'évolutivité.

1.8 Conclusion

Dans ce chapitre nous avons présenté les concepts de base des patterns du Gang Of four, on s'est focalisé en particulier sur les patterns Observer et Visitor. Dans le chapitre suivant on va présenter les concepts de base de l'ingénierie dirigée par les modèles et les transformations des modèles.

Chapitre 2

Transformation de modèles et Ingénierie Dirigées par les Modèles (IDM)

2.1 Pourquoi utiliser des modèles ?

Le modèle est une abstraction d'un système ou d'une partie de celui-ci. Selon le type de modèle, il peut fournir une vue simple du système ou plus détaillée. Dans les modèles de génie logiciel ont une longue et riche histoire. Les concepteurs utilisent souvent des modèles pour planifier leurs projets ou les expliquer à quelqu'un. Ces modèles sont normalement des esquisses simples qui ne contiennent aucune information pertinente sur la façon dont le projet doit être mis en œuvre. On prépare une présentation d'un projet on utilise des modèles un peu plus spécialisés. Aujourd'hui les modèles les plus couramment utilisés sont les modèles UML. Dans son livre, Martin Fowler explique trois utilisations de l'UML : UML as a Sketch, UML as a Blueprint et UML as a Programming language. Le dernier est devenu une possibilité avec le nouveau UML 2 dont nous parlerons bien plus tard, et aussi une bonne base pour utiliser le MDA. Aujourd'hui, un grand pourcentage des développeurs utilisent les modèles UML parallèlement aux systèmes logiciels qu'ils conçoivent. Il fournit un moyen efficace de communication entre les concepteurs, en particulier si les systèmes sont complexes et impliquent un certain nombre d'équipes dans la conception. Qu'est-ce qui serait plus facile à comprendre, des modèles graphiques ou des centaines ou des milliers de lignes de code ?

2.2 Introduction

Le développement d'applications logicielles de SI est puisque l'origine UML3 MDA est un concept pour le développement logiciel de SI basé sur la création de modèles et de transformations entre eux, défini par un organisme de normalisation en génie logiciel, OMG. OMG affirme que MDA est une étape évolutive dans le développement de logiciels. dans les années 90, lié à un niveau plus élevé d'abstraction. Beaucoup d'architectes de logiciels ont réalisé que le nombre de changements ; et édition dans le développement d'une application logicielle de l'IS a chuté quand nous examinons d'abord l'application sur un niveau plus élevé d'abstraction. Plus tard, ce fait est devenu un contexte pour la création de MDA [2]. MDA présente trois principaux avantages par rapport à d'autres méthodologies de développement logiciel : la transférabilité liée à l'indépendance de la plateforme, l'interopérabilité étroitement liée au développement stan-

dard et la réutilisabilité résultant des deux avantages précédents.

2.3 Transformation de modèle

La transformation de modèles consiste à passer d'un modèle source à un modèle cible , le passage de l'un de l'autre est décrit par des règles de transformation, Ces règles sont exécutées sur les modèles sources afin de produire les modèles cibles .
La figure suivante illustre le concept de la transformation modèle :

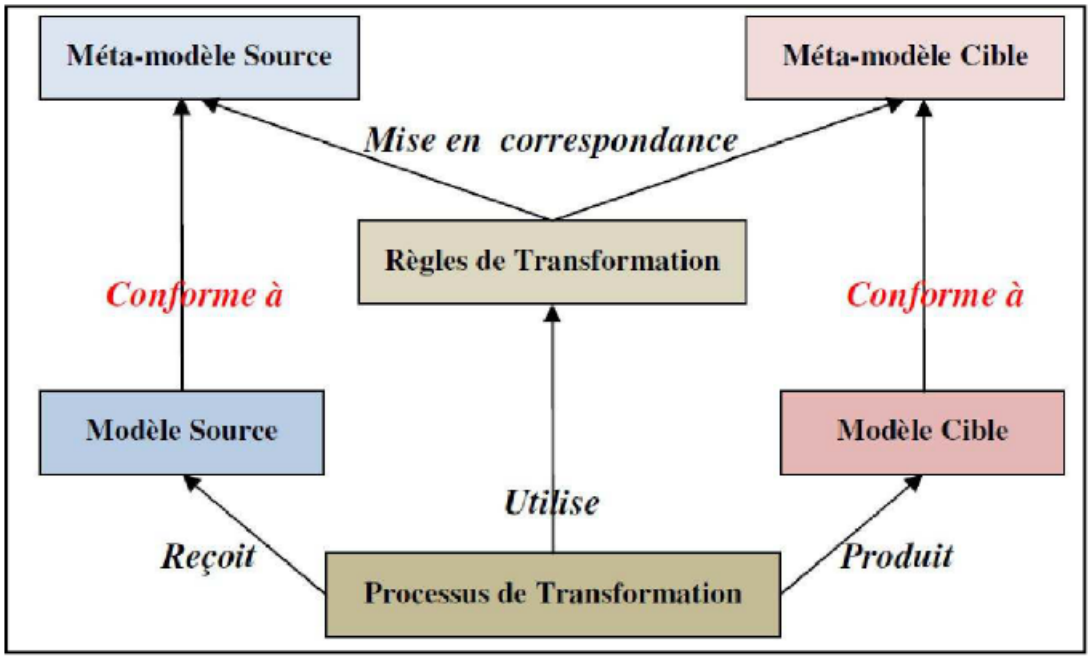

FIGURE 2.1 – Concept de la transformation modèle

La transformation des modèles est l'une des concepts les plus important de l'IDM .

2.3.1 Ingénierie dirigée par les modèles (IDM)

L'ingénierie Dirigée par les Modèles (IDM), ou Model Driven Engineering (MDE) est une discipline récente du génie logiciel qui met l'accent sur les modèles au sein du processus de développement logiciel, l'IDM

est donc le domaine de l'informatique mettant à disposition des outils, concepts et langages pour créer et transformer des modèles .

2.3.2 Concepts de base de l'IDM

- **Modèle :** c'est abstraction et simplification d'un système qu'il représente.
- **Un méta-modèle :** c'est d'un modèle qui définit le langage d'expression d'un modèle .
- **Méta-Méta-modèle :** permet de spécifier un langage de méta-modélisation pour exprimer les méta-modèles .

La relation entre un méta-méta-modèle et un méta-modèle est analogue à la relation pour exprimer les méta-modèles .
Ces concepts constituent la base d'un architecture à quatre niveaux adopté principalement par l'OMG illustré par la figure suivante :

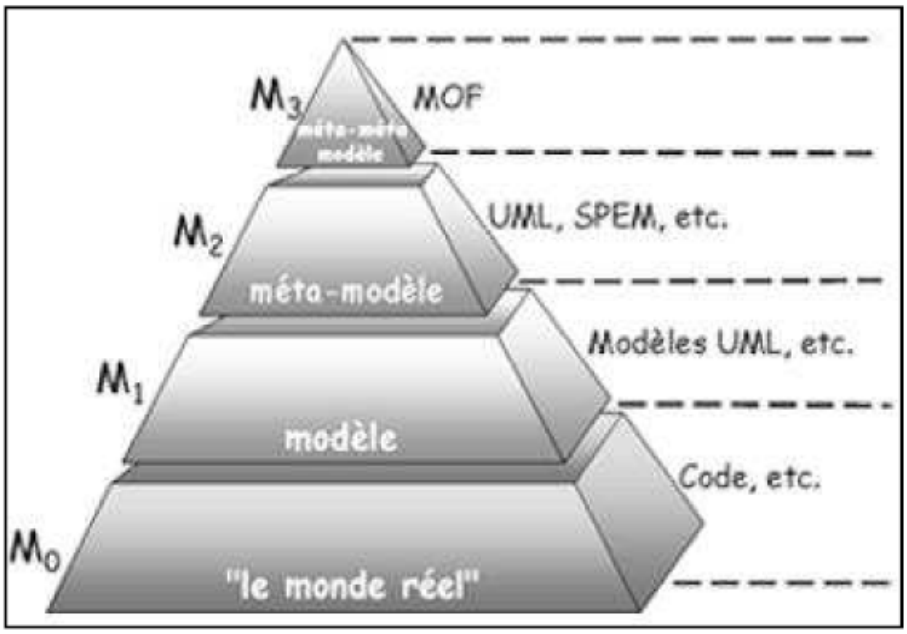

FIGURE 2.2 – Architecture à quatre niveaux de l'OMG

- Le niveau M0 : représente le modèle réel .
- Le niveau M1 : abstraction d'un système du monde réel .

- Le niveau M2 : ce sont les méta-modèles qui permettant la définition des ces modèles du M1
- Le niveau M3 : c'est le méta-méta-modèle, il représente le niveau le plus abstrait, il définit la structure de tous les méta-modèles du niveau M2 ainsi que lui meme .

Le méta-méta-modèle que nous allons utiliser dans notre transformation c'est le méta-méta-modèle Ecore introduit par EMF (Eclipse Modeling Framwork) décrit dans la figure suivante :

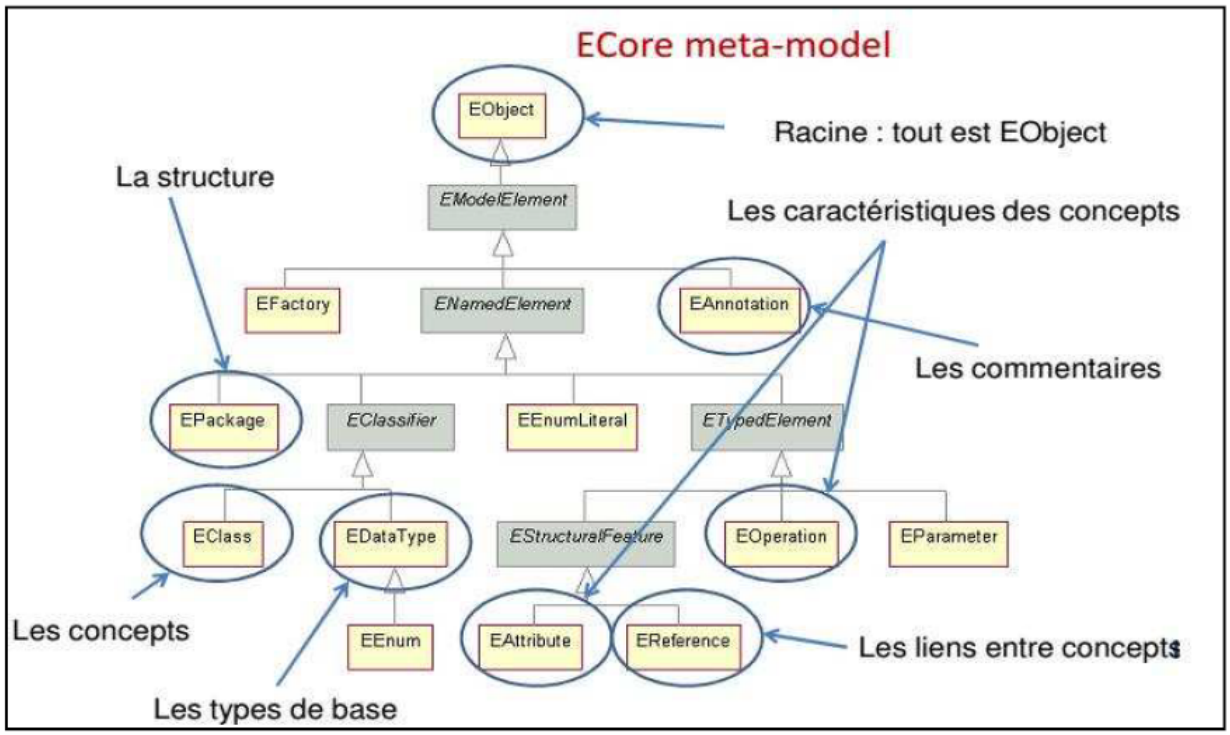

FIGURE 2.3 – Méta-méta-modèle Ecore

Durant le processus de transformation les modèles source et cibles doivent être conforme avec leur méta-modèle, les méta-modèles source et cible doivent aussi être conforme avec leur méta-méta-modèle qui est unique et conforme avec lui-même.

2.3.3 Classification des approches de transformations

Selon Czarnecki et Helsen on peut distinguer deux classes de transformation de modèles :
les transformations de type modèle vers code et les transformations de type modèle vers modèles, Généralement, le premier type de transformation peut être vu comme un cas particulier du deuxième type .

2.3.3.1 Une transformation de type modèle vers code :

Il existe deux approches de transformations de type modèle vers code :

- **a. les approches basées sur le principe du visiteur :** (Visitor-based approach) reposent sur le principe du visiteur, qui consistent å traverser le modèle en lui ajoutant des éléments (mécanismes visiteurs) et réduisent la différence de sémantique entre le modèle et le langage de programmation cible. Le code est obtenu en parcourant le modèle enrichi pour créer un flux de texte .

- **b. les approches basées sur le principe des patrons :** (Template-bascd approach) sont basées sur le principe des patrons sont actuellement les plus utilisées. Le code cible contient des morceaux de méta-code utilisés pour accéder aux informations du modèle source. La majorité des outils MDA couramment disponibles supporte ce principe de génération de code à partir de modèle. Parmi les outils basés sur ce principe, on peut citer : OptimalJ, XDE (qui fournissent la transformation modèle vers modèle aussi), JET, ArcStyler et AndroMDA (un générateur de code qui se repose notamment sur la technologie ouverte Velocity pour l'écriture des patrons)

2.3.3.2 Transformations de type modèle vers modèle :

Pour la classe de trans formation modèle vers modèle on distingue cinq approches :

- **a. Approches par manipulation directe :** Ces approches se basent sur une représentation interne des modèles source et cible, et sur un

ensemble d'APIS pour les manipuler. Elles sont généralement implémentées comme des cadres structurants orientés objets qui fournissent un ensemble minimal de concepts sous forme de classes abstraites par exemple. L'implémentation des règles et leur ordonnancement restent à la charge du développeur.

- **b. Approches relationnelles :** Ces approches sont celles qui utilisent une logique déclarative reposant sur des relations d'ordre mathématique. L'idée de base est de spécifier les relations entre les éléments des modèles source et cible par le biais de contraintes. L'utilisation de la programmation logique est particulièrement adaptée à ce type d'approche.
  Généralement, les transformations produites sont bidirectionnelles.

- **c. Approches basées sur les transformations de graphes :** Ces approches, qui exploitent les travaux réalisés sur les trans formations de graphes [Andries99]. Elles sont similaires aux approches relationnelles dans le sens où elles permettent l'expression des transformations sous une forme déclarative. Nčanmoins, les règles ne sont plus définies pour des éléments simples mais pour des fragments de modèles , on parle de filtrage de motif (pattern matching). Les motifs dans le modèle source, correspondant à certains critères, sont remplacés par 'd'autres motifs du modèle cible. Les motifs, ou fragments de modèles, sont exprimés soit dans les syntaxes concrètes respectives des modèles soit dans leur syntaxe abstraite.

- **d. Approches basées sur la structure :** Ces approches distinguent deux phases. La première consiste à créer la structure hiérarchique du modèle cible. La seconde consiste à ajuster les attributs et références dans le modèle cible.

- **e. Approches hybrides :** Les approches hybrides sont une combinaison des différentes techniques. On peut notamment retrouver des approches utilisant à la fois des règles à logique déclarative et des règles à logique impérative. ATL et XDE sont deux exemples d'approches hybrides.

2.3.4 Transformation d'UML vers les Modèles formels

Plusieurs travaux de transformation des diagrammes UML vers des modèles formelles ont été réalisés. L'objectif de ce domaine de recherche est de proposer une fondation formelle pour les modèle UML. Dupuy Sophie. Proposent de transformer les diagrammes de classes UML vers le langage Z par le biais des règles de traductions. Ouardani propose une transformation des diagrammes de séquence vers les réseaux de Petri sans considérer explicitement le temps, Ameedeen propose un autre travail qui transforme les diagrammes de séquence vers les réseaux de Petri dans le quel il s'intéresse au flux des événements décrit par le diagramme de séquence.

2.4 La transformation Model-To-Model avec ATL

Dans le domaine du génie logiciel, l'apparition de l'Ingénierie Dirigée par les modèles (IDM) propose une démarche dont les deux originalités sont d'une part la formulation des modèles, et d'autre part des programmes de transformation de modèles que nous allons y s'intéresser dans notre travail.

2.4.1 Model To Model « M2M »

- **M2M :** est la génération d'un ou plusieurs modèles cibles à partir d'un ou de plusieurs modèles sources.
- **Transformation :** ensemble de règles (correspondances entre éléments du modèle source et éléments du modèle cible).

2.4.2 Définition d'ATL

ATL est l'acronyme d'ATLAS Transformation Langage.

- Un langage de transformation de modèles dans le domaine de l'IDM ou MDE, développé par l'équipe de recherche ATLAS INRIA et LINA

- Un moyen de spécifier la manière de produire un certain nombre de modèles cibles à partir de modèles sources.
- Langage de transformation hybride (déclaratif et impératif).

2.4.3 Vue d'ensemble de l'approche de transformation ATL

ATL s'utilise dans le contexte de transformation présenté dans la figure suivante :

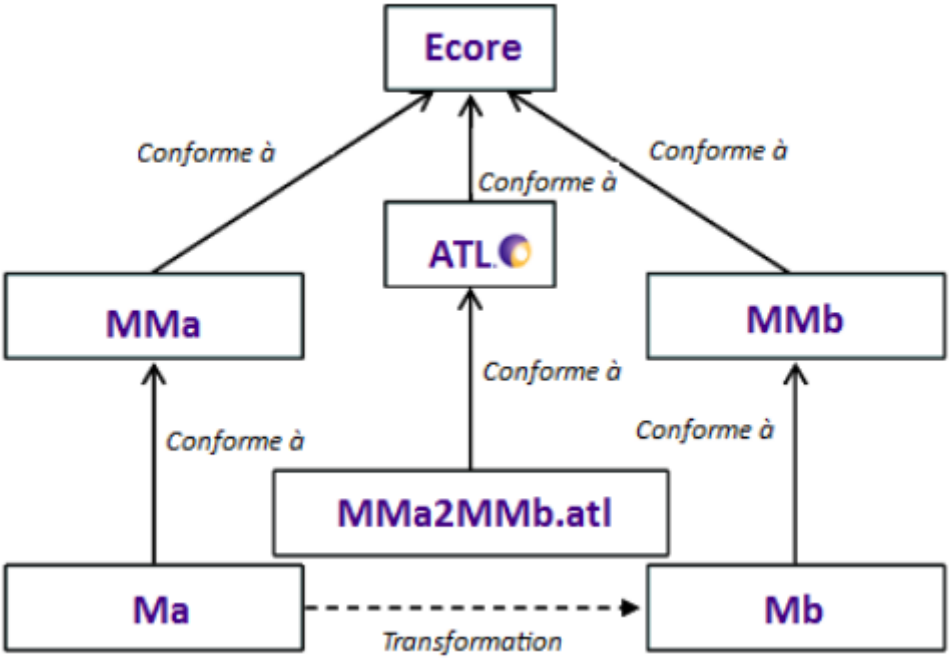

FIGURE 2.4 – Architecture d'ATL

Dans la figure 2.3, un modèle source Ma est transformé en un modèle cible Mb. La transformation est dirigée par un programme de transformation mma2mmb.atl écrit en ATL. Ce programme est un modèle. Les modèles source et cible ainsi que le programme de transformation sont conformes à leurs métamodèles respectifs :MMa, MMb et ATL. Ces méta-modèles sont conformes au méta-méta-modèle Ecore.

2.5 Présentation d'ATL

Dans cette partie, nous présentons les fonctionnalités du langage ATL .
En ATL, une transformation s'appelle un module.

2.5.1 Structure d'une transformation (module) :

- Déclaration du module
  - Header section
- Import de librairies
  - Import section
- Opérations
  - Helpers
- Règles de transformation
  - Rules

Header : donne le nom du module de transformation et déclare les modèles source et cible.
Exemple :

```
module MyRules;
create OUT : MM1 from IN : MM;
```

FIGURE 2.5 – Header

Import : sert à importer quelques bibliothèques ATL existantes.

Helpers : sont des fonctions ATL d'après le standard OCL sur le quel ATL se base. OCL définit deux sortes de helpers : opération et attribut.

- Exemple de helpers opération :

```
helper def :carre(x: Real): Real =x * x;
```

FIGURE 2.6 – helpers opération

- helpers attribut : sont des helpers sans paramètre.

Rules : Définissent la façon dont les modèles cibles sont générés a partir de modèles sources, on a deux types de règles : Matched rules et Called rules .

-Exemple de Matched rules :

```
rule ForExample {
  from
     i : InputMetaModel!InputElement
  to
     o : OutputMetaModel!OutputElement(
         attributeA <- i.attributeB,
         attributeB <- i.attributeC + i.attributeD
         )
 }
```

FIGURE 2.7 – Matched rules

Les requêtes ATL : Opération qui calcule une valeur primitive d'un ensemble de modèles de sources.

```
query query_name = exp;
```

FIGURE 2.8 – requêtes ATL

Les bibliothèques ATL : définir un ensemble de Helpers qui peuvent être appelées à partir des différentes unités ATL .

2.5.2 Les modes d'exécution des modules

Le moteur d'exécution ATL définit deux modes d'exécution pour les différents modules ATL :

- Le mode normal
- Le mode raffinage

Le mode normal

- spécifier la manière dont les éléments de modèle cible doivent être générés à partir des éléments du modèle source.
- Ce mode est spécifié par le mot clé **from** dans l'en-tête.
- Il est utilisé dans le cas d'une transformation **exogène** : le méta-modèle source et cible sont différents.

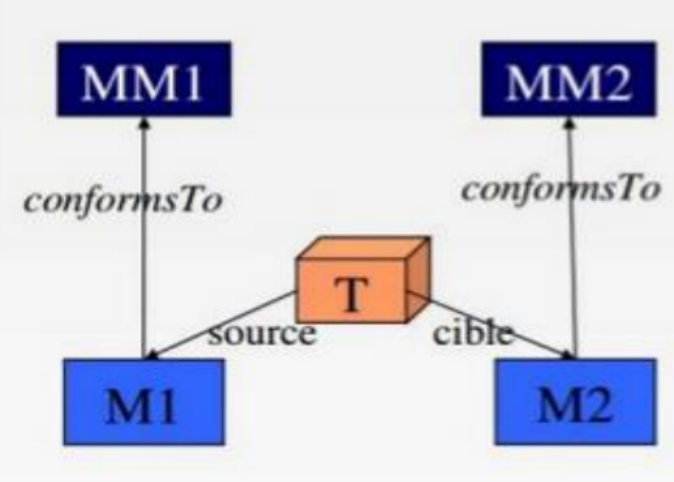

FIGURE 2.9 – transformation exogène

Le mode raffinage

- Un modèle M1 devient un modèle M2
- Ce mode est spécifié par le mot clef **refining** dans l'en-tête.
- Il est utilisé dans le cas d'une transformation **endogène** : un même méta-modèle source et cible

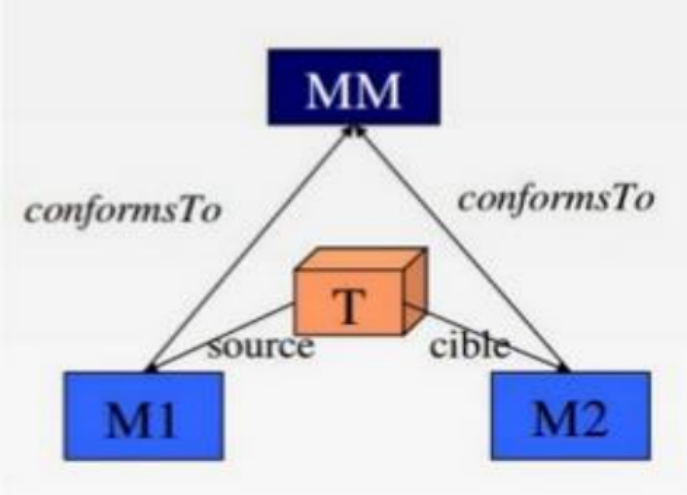

FIGURE 2.10 – transformation endogène

XMI « XML Metadata Interchange »

Les modèles et leurs méta-modèles cibles et source prennent dans l'environnement de développement Eclipse la forme de fichiers texte au format xmi avec une extension .ecore.
XMI permet de représenter n'importe quel modèle sous forme de document XML. Le principe de fonctionnement de XMI consiste à générer automatiquement une specification de structuration de balises à partir d'un méta-modèle.

2.6 Conclusion

Nous avons présentés les notions relatives aux transformations de modèles et ces différentes type , et ensuite on a présenté la transformation Model-To-model et nous avons expliqué son fonctionnement .

Dans le chapitre suivant nous verrons une composition de Pattern pour élaborer notre proposition et définir ainsi notre modèle de composition.

Chapitre 3

Les Patterns dans les Modèles de Transformation

3.1 Introduction

Notre objectif est de proposer une approche basée sur les patterns Observer et Visitor pour la prise en charge d'une transformation de modèles vers modèles. Notre modèle de conception utilise les métamodéles des langages de programmation source et cible (ex Java et XML). Notre modèle de conception propose un standard de développement des transformations de modéles indépendament des langages dédiées aux transformations ainsi nous offrons une souplesse de développement et une facilité d'implémentation. L'impléméntation du modèle sera la transformation d'un modèle ecrit sous Java qui sera transformé vers un fichier XML

Le résultat de cette transformation assure donc la conservation de l'aspect sémantique du modèle source.

L'utilisation de ce modèle de conception pour la transformation des modèles assure la facilité de la réutilisation des programmes et des données entre le langage source et cible

3.2 Combinaison de Patterns

Pour résoudre notre problème on a va utiliser les deux Patterns Observer et Visitor ensemble en même temps , ça veut dire on va combiner les deux Patterns d'une manière qu'ils vont fonctionner comme si ils sont un seul Pattern .

Le modèle proposé est illustré dans le schéma ci dessous et qui présente la combinaison des deux patterns pour la prise en charge de la transformation d'un modèle vers un autre modèle à notre que chaque modèle dépend de son métamodéle :

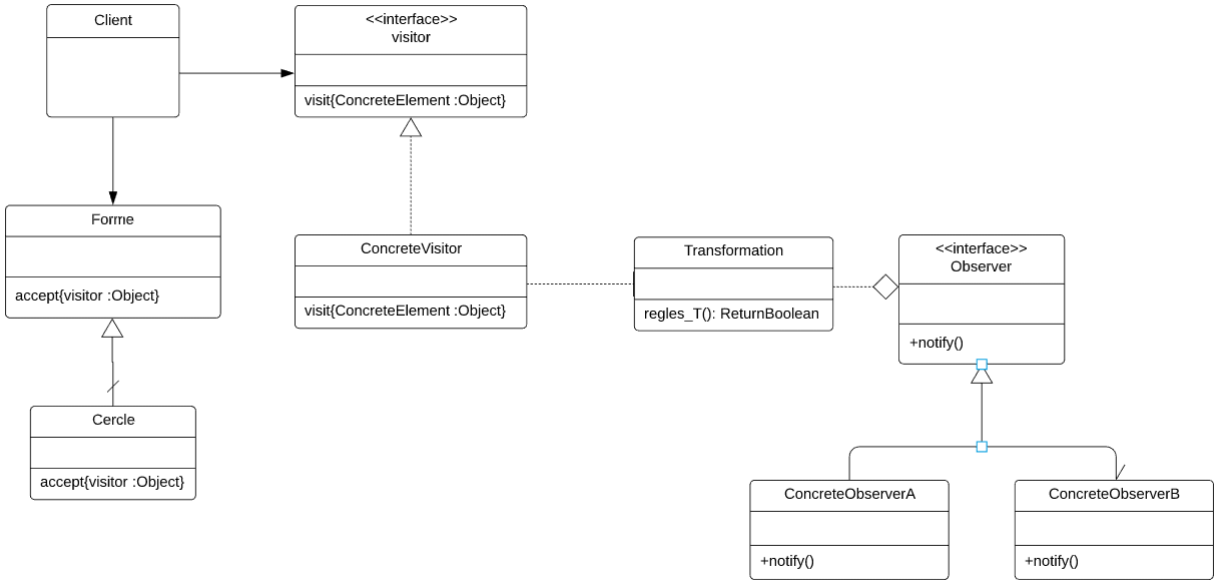

FIGURE 3.1 – Combinaison de les deux Patterns Observer et visitor

Implémentation

Le diagramme UML du notre Pattern définit une seule interface et six classes. Figure3.1

- Le visiteur déclare une méthode "visit", qui prend l'élément comme argument, pour chaque classe d'élément. Les visiteurs concrets sont dérivés de la classe de visiteur et implémentent ces méthodes "visit", chacune implémentant une partie de l'algorithme opérant sur la structure de l'objet. L'état de l'algorithme est maintenu localement par la classe de visiteur concrète.

- L' élément déclare une "accept" méthode pour accepter un visiteur, en prenant le visiteur comme argument. Les éléments concrets , dérivés de la classe d'élément, implémentent la méthode "accept" . Dans sa forme la plus simple, il ne s'agit que d'un appel à la "visit" du visiteur . Les éléments composites , qui gèrent une liste d'objets hérités, effectuent généralement des itérations sur ces derniers, appelant la méthode "accept" de chaque classe héritée .

- Le client crée la structure de l'objet, directement ou indirectement, et instancié les visiteurs concrets. Lorsqu'une opération doit être effectuée qui est implémentée à l'aide du modèle Visiteur, elle appelle la méthode "accept" du ou des éléments de niveau supérieur.

- Lorsque la méthode "accept" est appelée dans le programme, son implémentation est choisie en fonction à la fois du type dynamique de l'élément et du type statique du visiteur. Lorsque la méthode "visit" associée est appelée, son implémentation est choisie en fonction à la fois du type dynamique du visiteur et du type statique de l'élément, tel que connu dans l'implémentation de la méthode "accept", qui est le même que le type dynamique de l'élément.
- la classe "ConcreteVisitor" ne met pas à jour directement l'état des objets dépendants. Au lieu de cela, "ConcreteVisitor" fait référence à l' Observer (notify()) pour la mise à jour de l'état, ce qui rend "ConcreteVisitor" indépendant de la façon dont l'état des objets dépendants est mis à jour. Les classes ConcreteObserverA et ConcreteObserverB implémentent l' Observer en synchronisant leur état avec l'état du "ConcreteVisitor".
- La classe verification fait le role de notre adaptateur, elle reçoit qu'il y'a une nouvelle mise à jour de notre fichier XML générée par la classe "XMLexportVisitor". Ensuite elle envoie à son tour un evenement à la classe "Observer" pour lui prédire qu'un changement a été fait.

Dans notre cas , on va utiliser un example où on convertir un programme JAVA contient des formes géométriques vers un fichier XML qui contient les memes formes avec les memes paramètres .

donc , le schéma sera comme suivant :

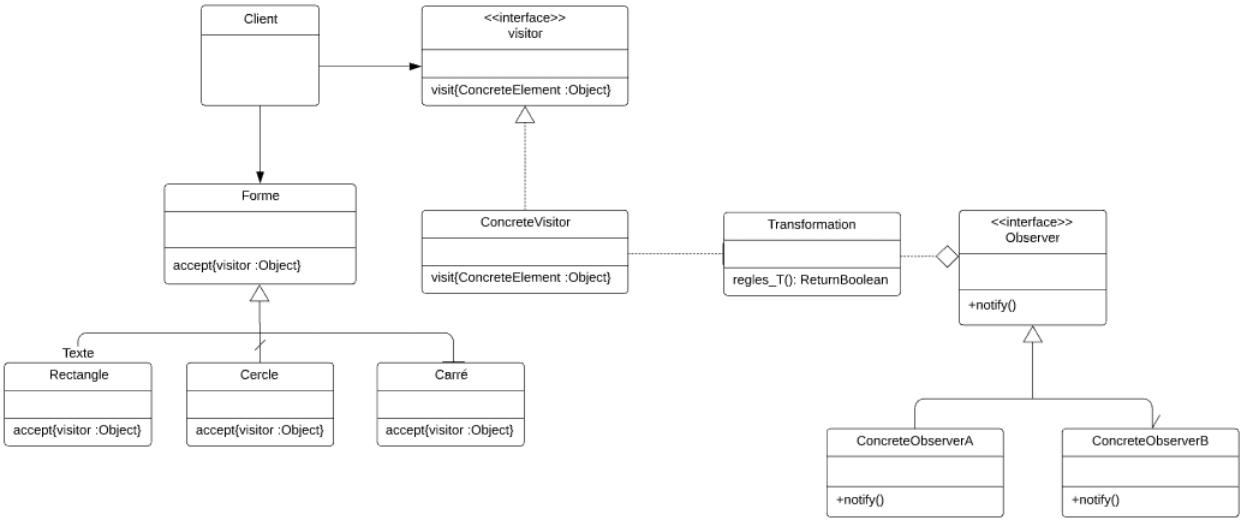

FIGURE 3.2 – Combinaison de les deux Patterns "notre cas"

3.3 Fonctionnement

Pour chaque modification au Modèle source (le code JAVA) , il sera un changement au Modèle cible (le fichier XML).

Lorsque un changement se fait , le Pattern Visitor se fonctionne premièrement , il va faire la mise à jour et appliquer le changement à le fichier XML généré , ensuite le Pattern Observer va relancer et notifier tout les destinations qu'on a un changement est appliqué et on a un nouveau fichier XML avec la dernier mise à jour .

Pour la transformation Model-To-Model, se fait quand on a un changement dans une règle de transformation dans le Modèle source. Donc pour ajuster et régler bien notre Pattern , et le rendre appliquer ces changement automatiquement au Modèle cible , on doit placer les méthodes de modification et les paramètres au meme place de les règles de transformation au niveau de la transformation Model-To-Model . Donc : les méthodes de modification et les paramètres de chaque forme ce sont notre règles de transformation .

.

le schéma suivant expliquer l'idée :

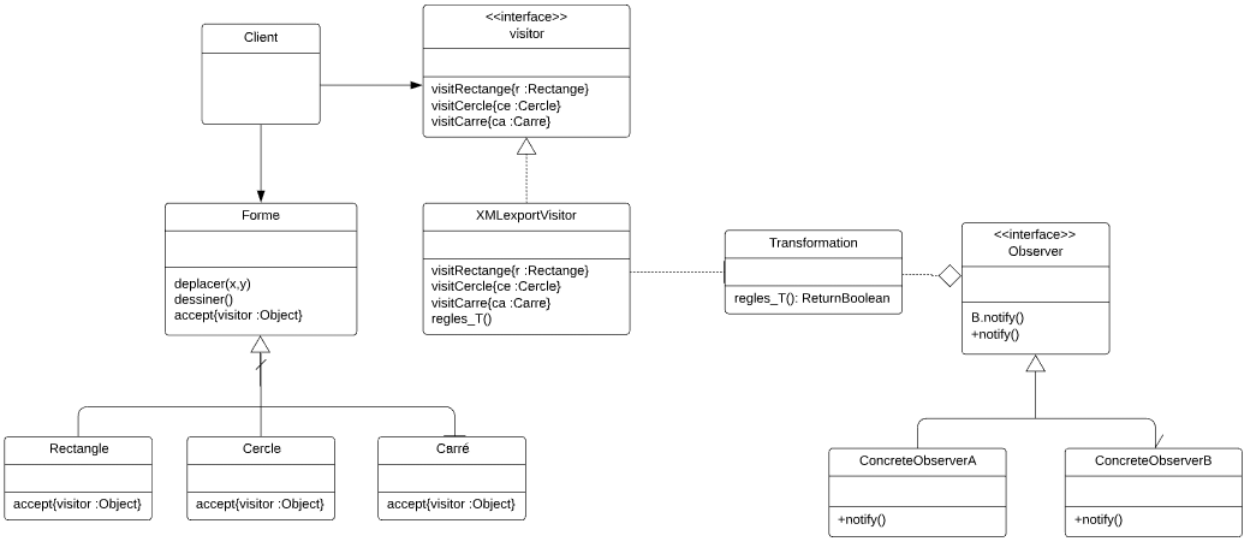

FIGURE 3.3 – règles de transformation

la transformation se fait sur notre modèle source l'une des classes JAVA qui représentent les différents type des formes qu'on a (rectangle,point,carré,triangle, cercle...), et les mêmes formes sont ainsi générés au niveau du modèle cible (fichier XML) avec les memes paramètres et caractéristiques .

La transformation Model-To-Model de notre projet se fait comme la Figure 3.4 explique :

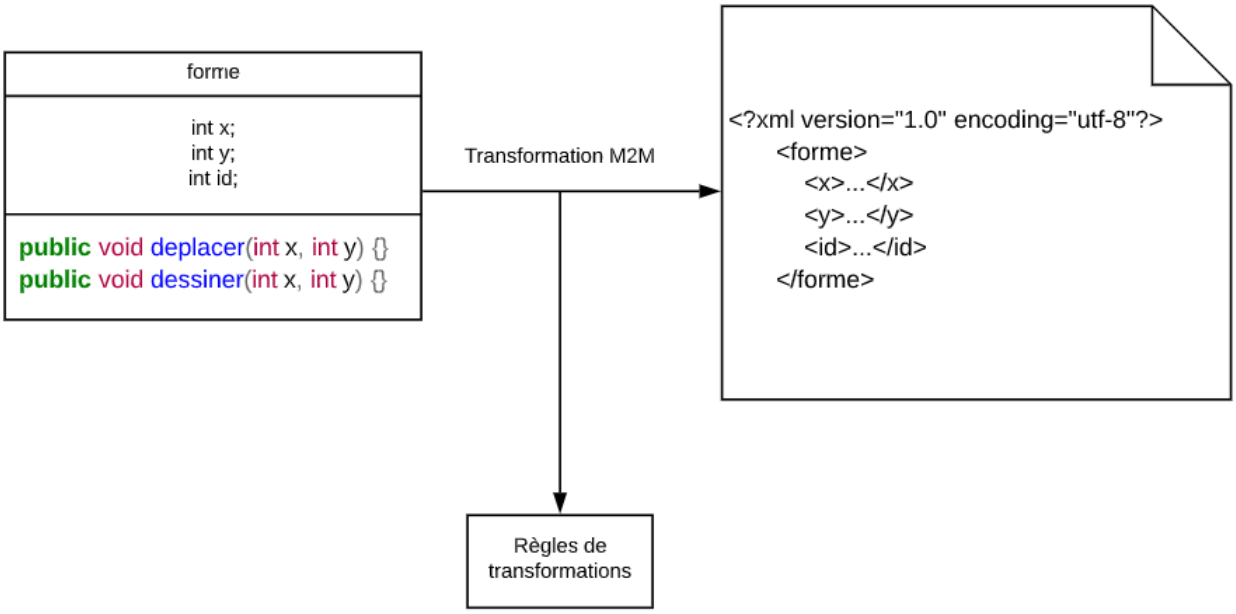

FIGURE 3.4 – notre transformation M2M

La classe "forme" représente le modèle source qui est notre modèle principale , ce modèle contient tout les paramètres de les formes différents .

Chaque forme a des paramètres différents aux autres formes. Le point est défini par les deux valeurs X et Y. Le cercle est défini par un point qui représente le centre de cercle et par la valeur de son rayon .Donc ,pour le cercle on a un paramètre de plus " R" et ainsi de suite pour les autres formes ...

Le ficher XML qui représente notre modèle cible qui est le résultat de la transformation Model-to-Model. Ce fichier contient les mêmes paramètres dans le modèle source, mais ils sont codés avec le langage XML.

example :

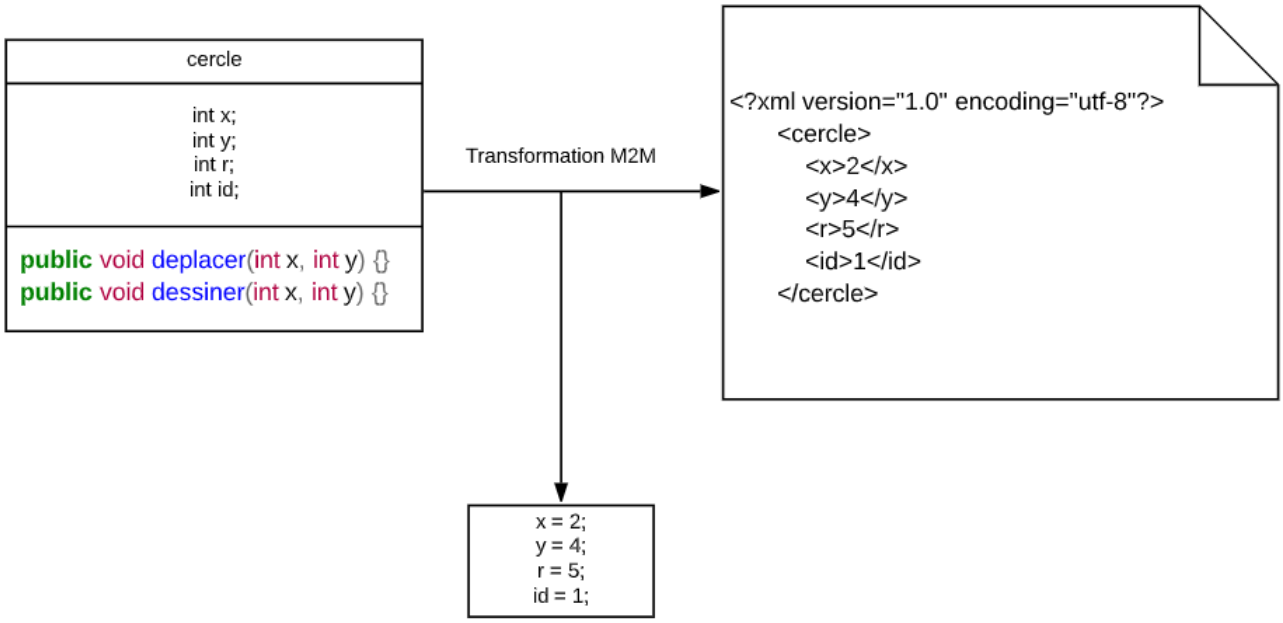

FIGURE 3.5 – exapmle cercle

Ensuite , un message va envoyer vers tout les destinations pour les notifier que on a un nouveau fichier XML, et ça c'est la dernière étape à faire pour notre programme.

Le schéma suivant, est une example donne une vision complète pour notre programme :

.

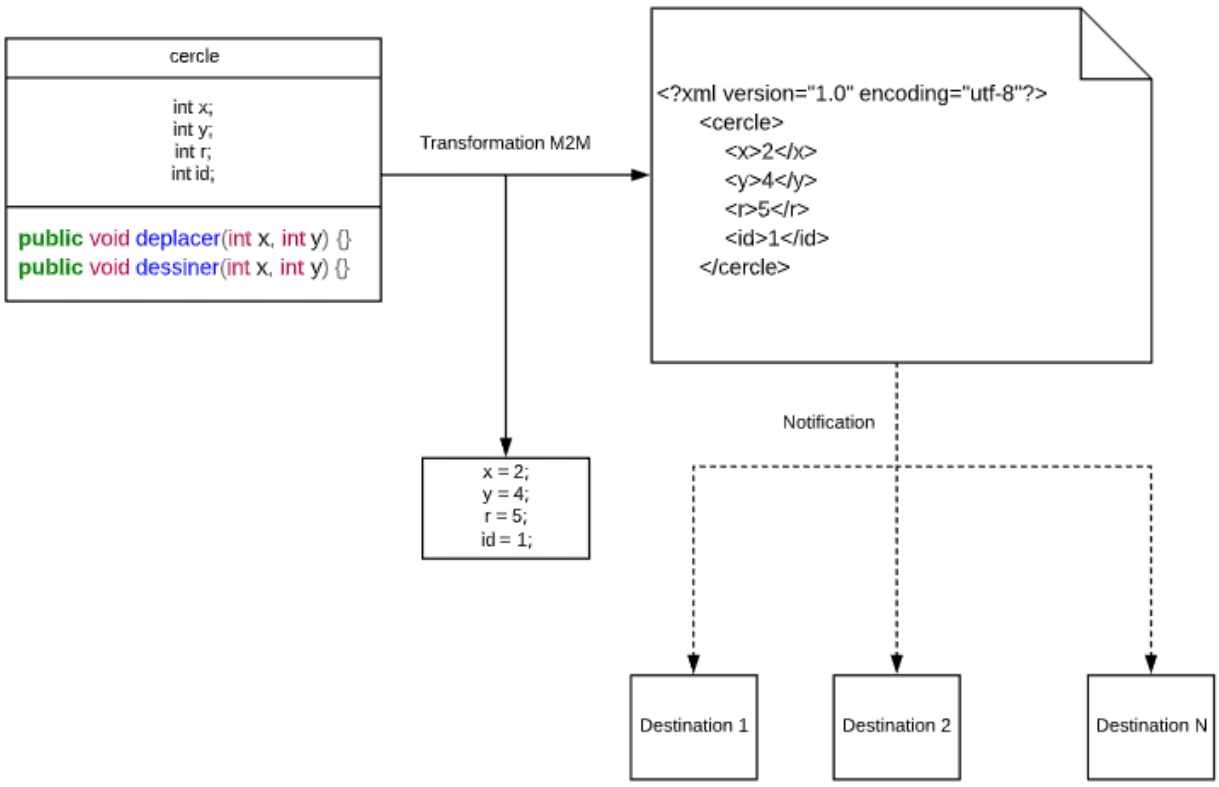

FIGURE 3.6 – notifier les destinations

3.4 Conclusion

Dans ce chapitre nous avons présenté comment on a combiné les Patterns Visitor et Observer dans un seul patterns , et utiliser le dans la transformations modèle (Model-To-model) , ensuite on a donné un example pour notre travail .

Dans la suite de ce mémoire on va présenté et valider ce travail par des exemples et des captures du résultat final.

Chapitre 4

Validation

4.1 Introduction

L'objectif de ce chapitre est de valider notre programme JavaToXML permettant de transformer une architecture logicielle décrite en JAVA vers un fichier XML valide .

Aussi on va donner un exemple et exécuter notre programme , et montrer des captures d'écran de le résultat final et de le code source .

Pour valider ce travail on a besoin de l'environnement ECLIPSE.

4.2 Exécution

Les classes de notre programme sont :

- point
- cercle
- rectange
- DetailForme
- visitorConcrete
- Transformation
- Observer
- Destination
- Demo (la classe Main)

Et on a deux interfaces :

- forme
- visitor

point.java

```
package refactoring_guru.visitor.example.shapes;
public class Point implements Forme {
    private int id;
    private int x;
    private int y;

    public Point() {
    }

    public Point(int id, int x, int y) {
        this.id = id;
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public int getId() {
        return id;
    }

    @Override
    public void move(int x, int y) {
        this.x = this.x + x;
        this.y = this.y + y;
    }

    @Override
    public void draw() {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public String accept(refactoring_guru.visitor.example.visitor.Visitor visitor) {
       return visitor.visitPoint(this);
    }
}
```

FIGURE 4.1 – la classe point

cercle.java

```
package refactoring_guru.visitor.example.shapes;

import refactoring_guru.visitor.example.visitor.Visitor;

public class Cercle extends Point {
    private int rayon;

    public Cercle(int id, int x, int y, int rayon) {
        super(id, x, y);
        this.rayon = rayon;
    }


    public String accept(Visitor visitor) {
        return visitor.visitCercle(this);
    }

    public int getRayon() {
        return rayon;
    }
}
```

FIGURE 4.2 – la classe cercle

rectangle.java

```
package refactoring_guru.visitor.example.shapes;
import refactoring_guru.visitor.example.visitor.Visitor;
public class Rectangle implements Forme {
    private int id;
    private int x;
    private int y;
    private int largeur;
    private int hauteur;

    public Rectangle(int id, int x, int y, int largeur, int hauteur) {
        this.id = id;
        this.x = x;
        this.y = y;
        this.largeur = largeur;
        this.hauteur = hauteur;
    }
   public int getId() {
        return id;
    }

    public int getX() {
        return x;
    }
```

```
    public int getY() {
        return y;
    }

    public int getLargeur() {
        return largeur;
    }

    public int getHauteur() {
        return hauteur;
    }

    @Override
    public void move(int x, int y) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public void draw() {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public String accept(Visitor visitor) {
        return visitor.visitRectangle(this);    }
}
```

FIGURE 4.3 – la classe rectangle

DetailForme.java

```java
package refactoring_guru.visitor.example.shapes;

import java.util.ArrayList;
import java.util.List;
import refactoring_guru.visitor.example.visitor.Visitor;

public class DetailsForme implements Forme {
    public int id;
    public List<Forme> fils = new ArrayList<>();

    public DetailsForme(int id) {
        this.id = id;
    }


    public void deplacer(int x, int y) {

    }


    public void dessiner() {

    }

    public int getId() {
        return id;
    }
```

```java
    public String accept(Visitor visitor) {
        return visitor.visitDetailsForme(this);
    }

    public void add(Forme f) {
        fils.add(f);
    }

    @Override
    public void move(int x, int y) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public void draw() {
        throw new UnsupportedOperationException("Not supported yet.");
    }

}
```

FIGURE 4.4 – la classe DetailForme

visitorConcrete.java

```
package refactoring_guru.visitor.example.visitor;

import refactoring_guru.visitor.example.shapes.Cercle;
import refactoring_guru.visitor.example.shapes.DetailsForme;
import refactoring_guru.visitor.example.shapes.Point;
import refactoring_guru.visitor.example.shapes.Forme;
import refactoring_guru.visitor.example.shapes.Rectangle;

public class XMLExportVisitor implements Visitor {

    public boolean b = false;

    public XMLExportVisitor() {
        this.b = false;

    }

    public String export(Forme... args) {
        StringBuilder sb = new StringBuilder();
       sb.append("<?xml version=\"1.0\" encoding=\"utf-8\"?>" + "\n");
        for (Forme f : args) {
            sb.append(f.accept(this)).append("\n");
            System.out.println(sb.toString());
            sb.setLength(0);
        }
        regles_T();
        return sb.toString();
    }
```

```
    @Override
    public String visitPoint(Point p) {
        return "<point>" + "\n" +
                "    <id>" + p.getId() + "</id>" + "\n" +
                "    <x>" + p.getX() + "</x>" + "\n" +
                "    <y>" + p.getY() + "</y>" + "\n" +
                "</point>";
    }


    @Override
    public String visitRectangle(Rectangle r) {
        return "<rectangle>" + "\n" +
                "    <id>" + r.getId() + "</id>" + "\n" +
                "    <x>" + r.getX() + "</x>" + "\n" +
                "    <y>" + r.getY() + "</y>" + "\n" +
                "    <largeur>" + r.getLargeur() + "</largeur>" + "\n" +
                "    <hauteur>" + r.getHauteur() + "</hauteur>" + "\n" +
                "</rectangle>";
    }

    @Override
    public String visitDetailsForme(DetailsForme df) {
        return "<DetailsForme>" + "\n" +
                "   <id>" + df.getId() + "</id>" + "\n" +
                _visitDetailsForme(df) +
                "</DetailsForme>";
    }
```

```
    private String _visitDetailsForme(DetailsForme df) {
        StringBuilder sb = new StringBuilder();
        for (Forme shape : df.fils) {
            String obj = shape.accept(this);
            obj = "    " + obj.replace("\n", "\n    ") + "\n";
            sb.append(obj);
        }
        return sb.toString();
    }

    @Override
    public String visitCercle(Cercle c) {
 return "<cercle>" + "\n" +
                    "    <id>" + c.getId() + "</id>" + "\n" +
                    "    <x>" +  c.getX() + "</x>" + "\n" +
                    "    <y>" +  c.getY() + "</y>" + "\n" +
                    "    <rayon>" + c.getRayon() + "</rayon>" + "\n" +
                    "</cercle>";    }


public void regles_T(){
    this.b=true;

}

}
```

FIGURE 4.5 – La classe visitorConcrete

Transformation.java

La classe "Transformation" fait la liaison entre les Patterns "Visitor" et "Observer", elle est notre Adaptateur, son rôle est de faire la combinaison entre les deux Patterns, après la fin de l'exécution du premier Pattern (Visitor) elle démarre automatiquement le deuxième Pattern (Observer) pour notifier toutes les destinations pour mettre à jour notre fichier XML.

```java
package Transformation;

import refactoring_guru.visitor.example.visitor.XMLExportVisitor;

public class Transformation {



    public boolean verifier(XMLExportVisitor s){

        return s.b;
    }

        public void notifyObserver(XMLExportVisitor s, Observer observer){

            if (verifier(s)){
                observer.notifyDestination();
            }

        }
}
```

FIGURE 4.6 – la classe Transformation

Observer.java

```java
package Transformation;

public class Observer {


    private Destination destination1 = new  Destination();
    private Destination destination2 = new  Destination();

    public void notifyDestination(){
       destination1.Notify();
       destination2.Notify();
     }
}
```

FIGURE 4.7 – la classe Observer

Destination.java

```
package Transformation;

public class Destination {
    private String name;


     public void Destination(String name){
       this.name = name;
    }

    public void Notify(){
        System.out.println("Le Fichier a été modifié");
    }
}
```

FIGURE 4.8 – la classe destination

Demo.java

```
package demo;
import Transformation.Observer;
import Transformation.Transformation;
import refactoring_guru.visitor.example.shapes.*;
import refactoring_guru.visitor.example.shapes.Point;
import refactoring_guru.visitor.example.visitor.XMLExportVisitor;
public class Demo {
        public static void main(String[] args) {
        Point p = new Point(1, 10, 55);
        Cercle c = new Cercle(2, 23, 15, 10);
        Rectangle r = new Rectangle(3, 10, 17, 20, 30);

        DetailsForme df = new DetailsForme(3);
        df.add(p);
        df.add(c);
        df.add(r);

        export( df);

    }

    private static void export(Forme... f) {

        Observer observer = new Observer();
        Transformation transformation = new Transformation();

        XMLExportVisitor exportVisitor = new XMLExportVisitor();
        System.out.println(exportVisitor.export(f));
        transformation.notifyObserver(exportVisitor ,observer );
    }
}
```

FIGURE 4.9 – la classe "Main" Demo

forme.interface

```
package refactoring_guru.visitor.example.shapes;

import refactoring_guru.visitor.example.visitor.Visitor;


public interface Forme {
    void move(int x, int y);
    void draw();
    String accept(Visitor visitor);
}
```

FIGURE 4.10 – l'interface forme

visitor.interface

```
package refactoring_guru.visitor.example.visitor;

import refactoring_guru.visitor.example.shapes.DetailsForme;
import refactoring_guru.visitor.example.shapes.Point;
import refactoring_guru.visitor.example.shapes.Cercle;
import refactoring_guru.visitor.example.shapes.Rectangle;


public interface Visitor {
    String visitPoint(Point p);

    String visitCercle(Cercle c);

    String visitRectangle(Rectangle r);

    String visitDetailsForme(DetailsForme df);
}
```

FIGURE 4.11 – l'interface visitor

Le résultat

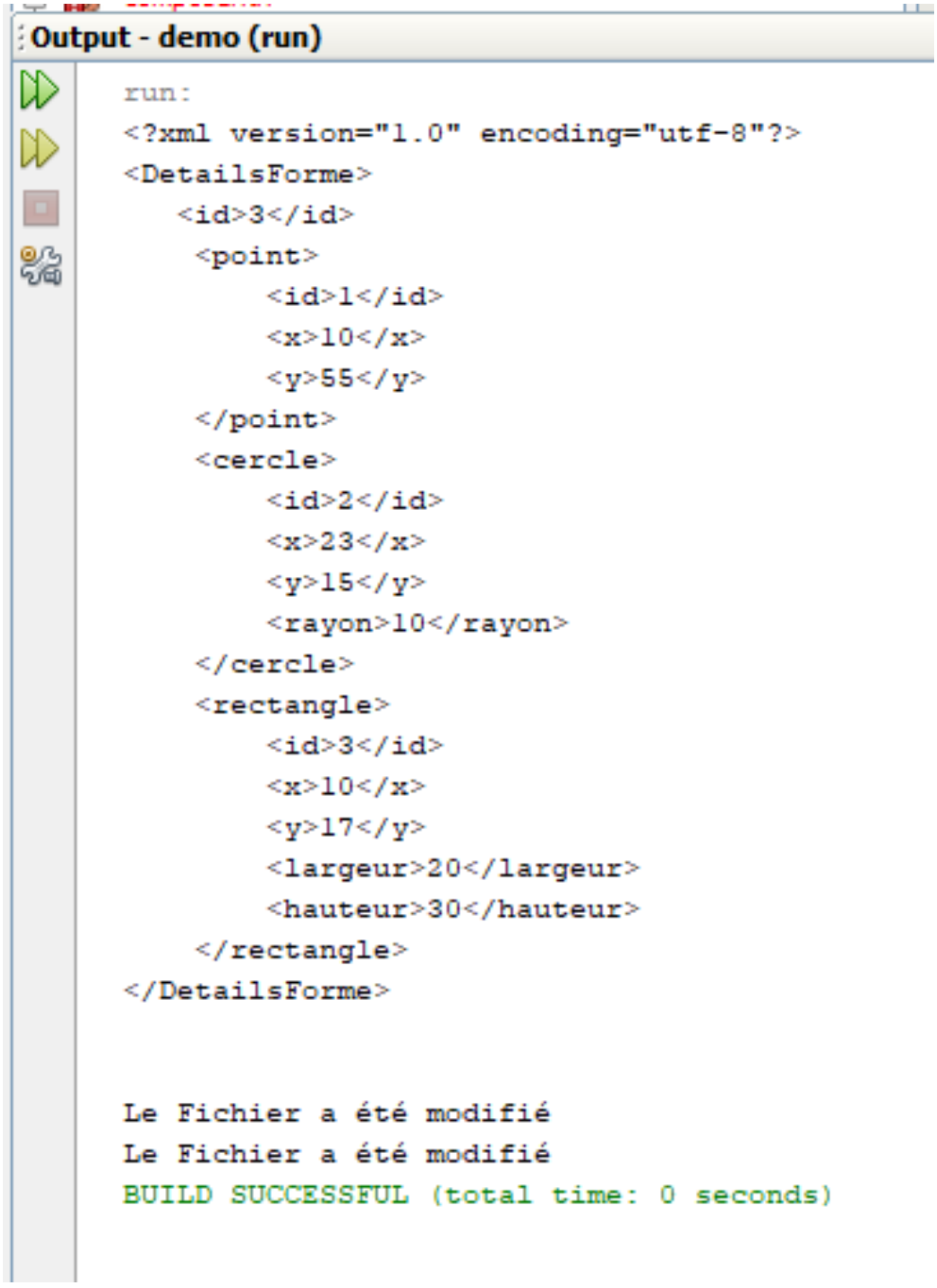

FIGURE 4.12 – résultat "fichier XML"

Conclusion Générale

Notre modèle de conception établis au chapitre 3 a permis une prise en considération la transformation des modèles en utilisant une composition de deux Pattern "Observer" et "Visitor" du gang of Four.
Une généralisation de ce modèle et son incorporation pour tous les langages de transformation de modèle tel que ATL, TGG et Atom3 sera d'une valeur intéressante.
L'intégration de ces Patterns comme un IDE sous eclipse est également un travail envisageable.

Bibliographie

[1] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. A Pattern Language : Towns, Buildings, Construction. Oxford University Press, 1977.

[2] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design Pattern Detection Using Similarity Scoring. IEEE Transactions on Software Engineering, 32(11) :896–909, November 2006.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns : Elements of Reusable Object-Oriented Software. Addison Wesley Professional, 1994.

[4] Sergio Antoy and Michael Hanus. New Functional Logic Design Patterns. In 20th International Workshop on Functional and Constraint Logic Programming, pages 19– 34, Odense, Denmark, July 2011.

[5] Anya Helene Bagge and Ralf Lammel. Walk Your Tree Any Way You Want. In 6th International Conference on Theory and Practice of Model Transformations, pages 33–49, Budapest, Hungary, June 2013.

[6] Ralf Lammel and Simon L. Peyton Jones. Scrap Your Boilerplate : A Practical Design Pattern for Generic Programming. In Proceedings of ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, pages 26–37, New Orleans, LA, January 2003.

[7] Ralf Lammel and Joost Visser. Design patterns for functional strategic programming. In Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-Based Programming, pages 1–14, 2002.

[8] Eelco Visser. A Survey of Rewriting Strategies in Program Transformation Systems. Electronic Notes in Theoretical Computer Science, 57 :109–143, 2001.

[9] Alexandre L. Correa, Claudia M. L. Werner, and Gerson Zaverucha. Object Oriented Design Expertise Reuse : An Approach Based on Heuristics, Design Patterns and Antipatterns. In Software Reuse : Advances in Software Reusability, volume 1844 of LNCS, pages 336–352, Vienna, Austria, June 2000.

[10] Brian Huston. The Effects of Design Pattern Application on Metric Scores. Journal of Systems and Software, 58(3) :261–269, 2001.

[11] Arlow, J ; Neustadt, I. UML2 and the Unified Process : Practical Object-oriented Analysis and Design, Addison Wesley, 2005

[12] OMG Inc. Model Driven Architecture (MDA) FAQ.... October, 29th 2009.

[13] Miller, J ; Mukerji, J. MDA Guide Version 1.0.1, OMG Inc., 2003.

[14] Bizoňová, Z ; Ranc, D. MDA used at analysis of LMS systems and creation of general model. In Objekty 2008 : 13th year of conference, Žilina, November, 20.21st, 2008 : Publisher : Edis, University of Žilina, pages 162-170, ISBN 978-808070-927-3, 2008.

[15] Kherraf, S ; Lefebvre, E ; Suryn, W. Transformation from CIM to PIM Using Patterns and Archetypes, 19th Australian Conference on Software Engineering, 2008.

[16] Rodríguez, A ; Fernández-Medina, E ; Piattini, M. Towards CIM to PIM Transformation : From Secure Business Processes Defined in BPMN to Use-Cases. In : Lecture Notes in Computer Science, Publisher : Springer Berlin / Heidelberg, ISBN 978-3-540-75182-3, pages 408-415, 2007.

[17] Zhang, W ; Mei, H ; Zhao, H ; Yang, J. Transformation from CIM to PIM : A Feature-Oriented Component-Based Approach. Work presented at MoDELS 2005, Montego Bay, Jamaica, 2005.

[18] QVT : Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, 2008,

[19] https ://www.javagists.com/adapter-design-pattern

[20] https ://fr.wikipedia.org/wiki/Observateur(patron-de-conseption)

[21] https ://en.wikipedia.org/wiki/Visitor-pattern