République Algérienne Démocratique et Populaire

Ministère de L'enseignement Supérieur et de la Recherche Scientifique

Université SAAD DAHLEB de BLIDA

Faculté Des Sciences

Département d'Informatique

MEMOIRE

Présenté pour l'obtention du diplôme de Master

Spécialité:

Génie Logiciel

Jury: Bousbia Narhimene

Réalisé par : MAHIEDDINE Mohamed

Thème

Etude, et réalisation des transformations de modèles M2M2T, à partir d'UML à un modèle de graphe, et du modèle de graphe en code Java, au moyen des patrons de conception.

Promoteur :

Dr MAHIEDDINE Mohammed.

Année universitaire 2011-2012.

REMECIEMENTS

Tout d'abord, je voudrais exprimer à Monsieur **Mohamed Mahieddine**, Chef de l'équipe Glodoo (LRDSI) , et Maître de conférences à l'université Saad Dahlab, mes vifs remerciements et ma profonde reconnaissance de m'avoir accueilli au sein de son équipe et encadré durant ma thèse, ainsi que pour sa disponibilité, ses orientations et ses conseils constants qu'il a pu émettre à la fois sur le fond et sur la forme de ce travail.

Je ne peux pas oublier de mentionner la grande compétence et la justesse d'appréciation de Monsieur Le Recteur, Monsieur Le Vice-recteur chargé de la post-graduation, et Monsieur Le vice doyen chargé de la post-graduation, concernant la gestion des problèmes des étudiants, et des conflits.

Que le conseil scientifique de la faculté des sciences, de par son directeur et tous ses membres acceptent ma profonde gratitude.

Je tiens à remercier vivement Monsieur , Professeur à l'université Saad Dahlab de Blida, pour l'honneur qu'il me fait en acceptant de présider le jury de ce mémoire.

Je saisis aussi cette opportunité pour exprimer mes remerciements et ma gratitude à tous ceux qui m'ont aidé de prés ou de loin et dont le soutien m'a été précieux durant la préparation de ce mémoire.

Enfin et profitant de cette occasion, je voudrais exprimer mes remerciements et ma sympathie à mes frère Youcef, Naima et Riadh , mes parents, mes grands parents « lah yarhamhom », et particulièrement à ma mère et à mon père qui ont accepté de me soutenir sentimentalement et matériellement pendant toute la durée de mes études, et particulièrement pendant le déroulement de cette thèse.

Résumé

Traditionnellement, les modèles sont considérés comme de jolies images qui servent seulement comme soutien dans la documentation d'un projet de développement logiciel.

Avec l'avenue du modèle de l'ingénierie dirigée par (MDE) ce point de vue doit être reconsidéré. Les Modèles sont considérés comme des artefacts de première classe qui sont à la base dela génération d'un code de programme exécutable.

Le Model-Driven Development (MDD) vise à remplacer les méthodes manuelles de développement de logiciels par des méthodes automatisées pour exprimer des concepts de domaine de manière efficace.

L'Ingénierie Dirigée par les Modèles (IDM) est une discipline récente du génie du logiciel qui met les modèles au premier plan au sein du processus du développement logiciel.

Elle a apporté plusieurs améliorations significatives dans le développement des systèmes logiciels complexes en fournissant des moyens permettant de passer d'un niveau d'abstraction à un autre ou d'un espace de modélisation à un autre.

Cependant, la gestion des modèles peut s'avérer lourde et coûteuse.
Pour pouvoir mieux répondre aux attentes des utilisateurs, il est nécessaire de fournir des outils flexibles et fiables pour la gestion automatique des modèles ainsi que des langages dédiés pour leurs transformations.

Dans ce mémoire, nous proposons un tour d'horizon sur les travaux récents de l'IDM en mettant l'accent sur la transformation de modèles qui constitue le thème central de cette discipline.

Nous allons utiliser ensuite les patrons de conception pour concevoir la double transformation de modèles M2M2C, qui transforme des modèles UML, d'abord en modèle de graphe, puis transforme ce dernier en modèle de code java.

TABLE DES MATIERES

TABLE DES FIGURES

CHAPITRE 1:

Introduction

Introduction

Le développement dirigé par les modèles ou Model-Driven Development (MDD) est un paradigme pour l'écriture et la mise en œuvre des programmes informatiques rapidement, efficacement et à moindre coût. Cette méthodologie est également connue comme l'ingénierie de développement de logiciels dirigée par les modèles (IDM).

L'approche MDD au développement de logiciels permet de réaliser des systèmes à partir de la transformation d'un modèle en un autre, jusqu'au modèle de code ou programme.

Le MDD permet aussi aux développeurs de travailler ensembles sur un même projet, même si leurs niveaux d'expériences individuelles varient grandement.

Il permet aux entreprises de maximiser le travail efficace sur un projet tout en minimisant les frais généraux nécessaires pour produire des logiciels de travail qui peuvent être validés par les utilisateurs finaux dans le délai le plus court possible.

Le Model Driven Development, en collaboration avec des outils associés basés sur UML, a vu le jour depuis plus d'une décennie maintenant [1].

Plusieurs organisations matures, techniquement compétentes, et avancées, tels que celles qui travaillent sur le projets 3G d'Ericsson [33] ont utilisé le MDD avec succès, ce qui a permis d'augmenter sensiblement leur avantage concurrentiel par des gains de marché suite à amélioration de la productivité, de la qualité et de la rapidité d'accès au marché.

Des études [5] révèlent que la productivité peut s'améliorer par un facteur de deux à quatre en utilisant le MDD à grande échelle, par rapport à l'élaboration axée ou centrée sur le développement du code de manière traditionnelle.

A l'avenir de nouveaux développements et extensions aux techniques actuelles de MDD vont voir le jour et promettent de continuer à stimuler l'avantage concurrentiel pour les organisations utilisant le MDD à grande échelle [2].

Aujourd'hui, les systèmes sont souvent de grande taille, si nous les mesurons par le nombre de classes, de cas d'utilisation, des lignes de code, de développeurs, de variantes, ou autre chose. Bien que la taille en elle-même ne se traduise pas nécessairement par une augmentation de la complexité, les systèmes actuels ont tendance à être beaucoup plus complexes que les précédents. Le coût de développement de ces systèmes est très élevé. Souvent, il n'est pas économiquement faisable de remplacer un système par un tout nouveau quand quelque chose change, par exemple si une nouvelle plateforme avec une meilleure capacité devient disponible ou d'autres exigences fonctionnelles apparaissent. Donc les systèmes actuels doivent être construits d'une manière qui permet des extensions futures, des modifications, et des portages. Cela exige une architecture de systèmes modulaire robuste et résistante, facilement accessible et compréhensible pour tout le personnel de développement. L'architecture doit également pouvoir être mise à jour au cours de la durée de vie du système pour être d'une quelconque aide pour les travaux futurs [6].

2. Problématique et objectifs

Dans le moment actuelle il ya plusieurs des logiciels de MDD qui permet de passer automatiquement d'un modèle a un autre.

Mais malheureusement, dans la plupart des outils de modélisation disponibles aujourd'hui [2], les modèles de génération de code sont généralement produits par la transformation entre des modèle particuliers uniquement, et aussi sont codés dans l'outil lui-même, et ne fournissent donc qu'un soutien très limité pour la personnalisation du code cible qui doit être généré.

Ce qui est demandé dans notre projet c'est de réaliser une application de MDD qui permet de faire le passage depuis le modèle UML au modèle de Graphes parce que ce modèle n'a pas encore été utilisé par les logiciels de MDD qui existent déjà sur le marché et de transformer ensuite le modèle de graphes en un modèle de code source, en l'occurrence java dans notre cas, et mettre ensuite le logiciel en open source pour permettre à l'avenir de passer directement du modèle réel en un modèle de graphe et ensuite, comme nous l'avons fait, de transformer le modèle de graphes en d'autre modèles selon les besoins.

Et pour cela on a utilisé trois modèles nécessaires dans notre application de MDD :

- le modèle UML,
- le modèle de GRAPHE et
- le modèle de code JAVA

et une double transformation :

- de UML au modèle de graphes, et
- du modèle de graphes au modèle de code JAVA

On va expliquer pourquoi on a choisi particulièrement ces trois modèles et pas d'autres, et pourquoi on a commencé par le modèle UML, et ensuite le modèle de GRAPHE et en dernier par le modèle de code JAVA.

Le choix de ses modèles nous a été dicté par :

Cas UML :

- UML est visuel et, s'il est utilisé correctement, et fonctionne à un niveau supérieur d'abstraction que celui du texte ou code machine,
- les développeurs peuvent se concentrer sur la conception logique plutôt que sur les questions de mise en œuvre, à un degré beaucoup plus élevé.

Cas Graphes :

On a utilisé le modèle de graphes qui est un modèle très puissant (très représentatif) et riche en opérateurs, et très utilisé par les développeurs.

Et ce dernier preuve leur importance jour après jour d'après les résultats obtenue

dans les différentes Domain de développement.

Et donc l'utilisation de ce dernier facilite au chercheur qui utilise les modèles des graphes dans leur développement de faire le passage automatique depuis leur model au autre modèle de facilement.

Cas JAVA :

On a utilisé le modèle de code Java parce que le langage JAVA possède de nombreux avantages non négligeables par rapport à d'autres langages de programmation orientés objets, comme par exemple :

- Portabilité excellente,
- Langage puissant,
- Langage orienté objet, supporté par de nombreuses entreprises telles que Sun ou encore IBM et des projets comme Apache…

CHAPITRE 2:

Les Notions Graphiques

Les Notions Graphiques

1. L'outil de Développement

1.1. Programmation Orientée Objets

Certains des avantages de l'utilisation de la programmation orientée objets sont la compréhensibilité de la conception, l'encapsulation de l'information, l'extensibilité et la réutilisation du programme pour un futur développement, et la modularité de la conception.

Ce qui rendra le programme facile à examiner, à tester et à retrouver d'éventuelles erreurs [3].

1.1.1. Le Paradigme Orienté Objet

Le paradigme orienté objet se concentre sur les caractéristiques comportementales et structurales des entités en tant qu'unités complètes.

Il est centré sur le concept (holistique) parce qu'il se concentre sur tous les types de dispositifs qui constituent n'importe quel concept donné **[24]**.

Le paradigme englobe et comporte les piliers suivants (premiers principes) :

- *L'abstraction* implique la formulation des représentations en se concentrant sur des similitudes et des différences parmi un ensemble d'entités pour extraire des caractéristiques essentielles de qualité intrinsèque (dispositifs communs appropriés) et pour écarter des caractéristiques fortuites extrinsèques (dispositifs de distinction non pertinents) afin de définir une représentation simple ayant ces caractéristiques qui sont appropriées à définir chaque élément dans l'ensemble.
- *L'encapsulation* comporte l'empaquetage des représentations en se concentrant sur se cacher des détails pour faciliter l'abstraction, où *des caractéristiques* sont employées pour décrire ce qu'une entité est et quelle entité fait, et *des réalisations* sont employées pour décrire comment une entité est réalisée.
- *L'héritage* comporte la relation et la réutilisation des représentations existantes pour définir de nouvelles représentations.
- *Le polymorphisme* implique la capacité de nouvelles représentations d'être définies comme variations des représentations existantes, où de nouvelles réalisations sont présentées mais leurs caractéristiques demeurent les mêmes dans le cas où les spécifications peuvent avoir plusieurs réalisations.

Ces piliers sont employés pour faciliter la communication, améliorer la productivité et l'uniformité, et de permettre la gestion du changement et de la complexité dans des efforts de résolution des problèmes **[24].**

1.1.2. Principes de la Programmation Orientée Objets [6]

1.1.2.1. Minimiser l'accessibilité des classes et de leurs membres

L'abstraction est le moyen fondamental pour maîtriser la complexité.

Une abstraction se concentre sur la vue externe d'un objet et sépare le comportement d'un objet de son implémentation.

C'est pourquoi les classes devraient être opaques (ne pas exposer les détails internes de leur implémentation).

Ce principe est connu sous le nom d'encapsulation et est une des bases de la programmation orientée objet. L'encapsulation rend la modification du comportement interne d'un objet transparent pour les autres objets, et rend la conception plus simple, puisque chaque objet contrôle l'accès à ses propres données **[24]**.

1.1.2.2. Favoriser la composition d'objets par rapport à l'héritage des classes

Les principales façons utilisées pour réutiliser le comportement sont

- L'héritage, et
- La composition

Dans l'héritage

- il est plus facile de redéfinir les méthodes héritées, et
- le code est statique (plus simple à suivre).

La composition

- comporte moins de programmation, et
- le changement du comportement s'effectue en temps d'exécution

La composition est une méthode de réutilisation des classes existantes qui consiste à créer un objet composé d'autres objets. La nouvelle fonctionnalité est obtenue en déléguant les requêtes aux objets entrant dans la composition.

C'est en règle générale une meilleure technique que l'ajout de nouvelles fonctionnalités par héritage, ce qui expose les détails internes des classes parent aux sous-classes, et ainsi viole le principe d'encapsulation. Au contraire, la composition d'objets est une réutilisation en tant que *boîte noir*", puisque les détails internes des objets contenus ne sont pas visibles.

Un autre inconvénient de l'héritage de classe est que celui-ci est défini statiquement au moment de la compilation, alors que la composition peut être définie dynamiquement lors de l'exécution, ce qui résulte en un système plus flexible.

Toutefois, l'héritage n'est pas en soi une mauvaise technique, mais il a été utilisé de manière automatique en tant que méthode de réutilisation, quand la composition d'objets mène souvent à une conception plus souple et plus simple **[24].**

UML

2. UML

2.1. Définition :

Est un langage de modélisation unifié (Unified Modeling Language)

Langage = syntaxe + sémantique :

Syntaxe : notations graphiques consistant essentiellement en des représentations conceptuelles d'un système.

Sémantique : sens précis pour chaque notation.

UML est un langage *(et non pas une méthode)* qui :

- permet de représenter les modèles.
- ne définit pas le processus d'élaboration des modèles **[10], [12]** .

2.2. Caractéristiques d'UML:

- un travail d'expert

- utilise l'approche orientée objet
- normalisé, riche
- Formel : sa notation limite les ambiguïtés et les incompréhensions
- langage ouvert
  - ***INDÉPENDANT*** du langage de programmation,
  - Domaine d'application : permet de modéliser n'importe quel système **[10], [11], [12]**.

2.3. Les Diagramme UML

UML 1.1 comprend **9** de diagrammes :
UML définit deux types de diagrammes, structurels (statiques) et comportementaux (dynamiques).

- **Modélisation de la structure**
  - diagramme de classes
  - diagramme d'objets
  - diagramme de composants
  - diagramme de déploiement

- **Modélisation du comportement**
  - diagramme de cas d'utilisation
  - diagramme d'états
  - diagramme d'activités
  - diagramme de collaboration
  - diagramme de séquence

Les diagrammes d'UML peuvent être utilisés pour représenter différents points de vue :

- **Vue externe** : vue du système par ses utilisateurs finaux
- **Vue logique statique** : structure des objets et leurs relations
- **Vue logique dynamique** : comportement du système
- **Vue d'implémentation** : composants logiciels
- **Vue de déploiement** : répartition des composants

2.3.1. Diagramme de classes

Le diagramme de classes constitue un élément très important de la modélisation : il permet de définir quelles seront les composantes du système final : il ne permet en revanche pas de définir le nombre et l'état des instances individuelles.
Néanmoins, on constate souvent qu'un diagramme de classes proprement réalisé permet de structurer le travail de développement de manière très efficace, il permet aussi, dans le cas de travaux réalisés en groupe (ce qui est pratiquement toujours le cas dans les milieux industriels), de séparer les composantes de manière à pouvoir répartir le travail de développement entre les membres du groupe.
Enfin, il permet de construire le système de manière correcte (***Build the system right***) **[10]**, **[11]**.

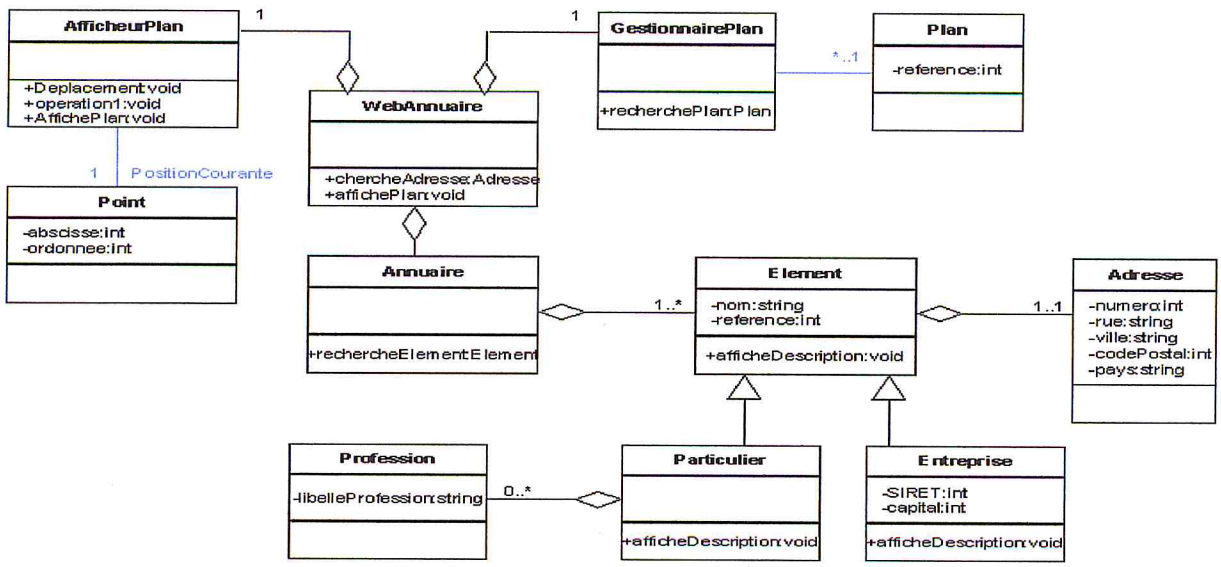


Figure 01 : *Représentation graphique d'un diagramme de classe*

- **Permet de donner une vue statique du système en terme de :**
  - Classes d'objets
  - Relations entre classes
    - ✓ Associations
    - ✓ agrégation/composition
    - ✓ héritage

- **La description du diagramme de classes est centrée sur trois concepts :**
  - Le concept d'objets
  - Le concept de classes d'objets comprenant des attributs et des opérations
  - Les différents types de relations entre classes.

2.3.1.1. Les Classes

La notion de classe est essentielle en programmation orientée objets : elle définit une abstraction, un type abstrait qui permettra plus tard d'instancier des objets. On distingue généralement entre classes abstraites (qui ne peuvent pas être instanciées) et classes "normales", qui servent à définir des objets **[10], [11]**.

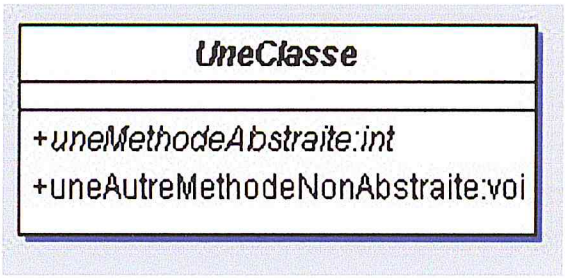


Figure 02: *Représentation graphique d'une Classe*

2.3.1.2. Les Relations Entre Les Classes

Les diverses classes possèdent des relations de dépendance entre elles. Ces relations possèdent en principe un équivalent syntaxique dans le langage de projection.
Les principales de ces relations sont énumérées dans la suite. Il est néanmoins important de noter que certaines relations peuvent ne pas avoir d'équivalent dans le langage de projection considéré.
Ainsi, C++ introduit la notion de template, ou classe paramétrable, qui peut être modélisée en UML, mais qui représente une notion inconnue en tant que telle en Java.
A l'inverse, Java permet de définir une classe qui implémente une interface, alors que la notion d'interface est inexistante en C++. Dans les deux cas, l'outil de modélisation, s'il génère du code, choisira le mode de représentation le mieux adapté en considération du langage de projection choisi.
Une interface UML pourrait, par exemple, se traduire par une classe abstraite en C++ **[10], [11]** .

2.3.1.2.1. Agrégation

Type particulier d'association dans laquelle :

- Classe agrégat (composé), classes agrégée (composant)
  Entre les deux, il existe une relation de type « est composé de »

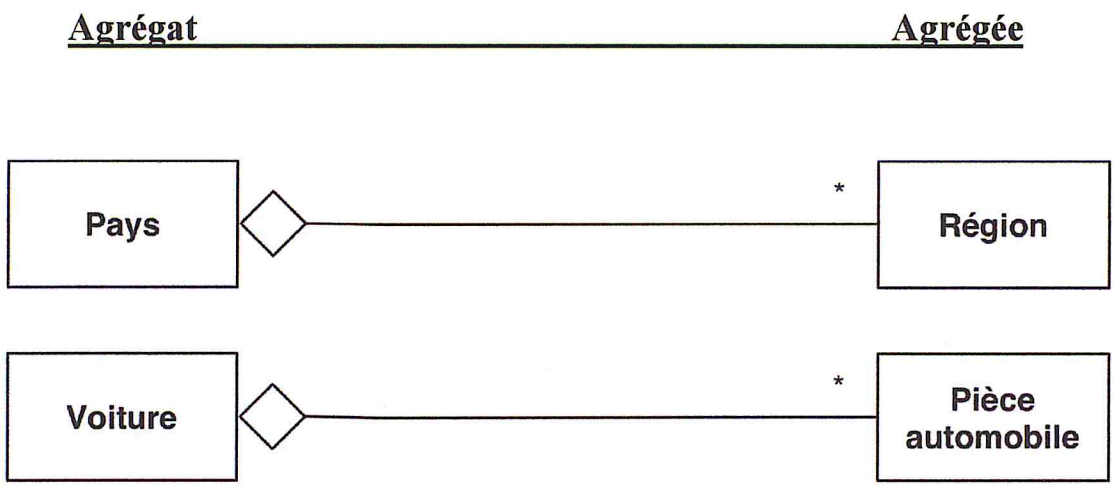


Figure 03 : *représentation graphique d'une Agrégation*

2.3.1.2.2. La Composition

- La composition est un cas particulier d'une agrégation dans laquelle la vie des composants (élément) est liée à celle de l'agrégat (composé) : si l'agrégat est détruit (ou déplacé), ses composants le sont aussi.
- D'un autre côté, et contrairement à l'agrégation, une instance de composant ne peut être liée qu'a un seul agrégat.
- La composition se représente par un losange noir (plein) **[10], [11]**.

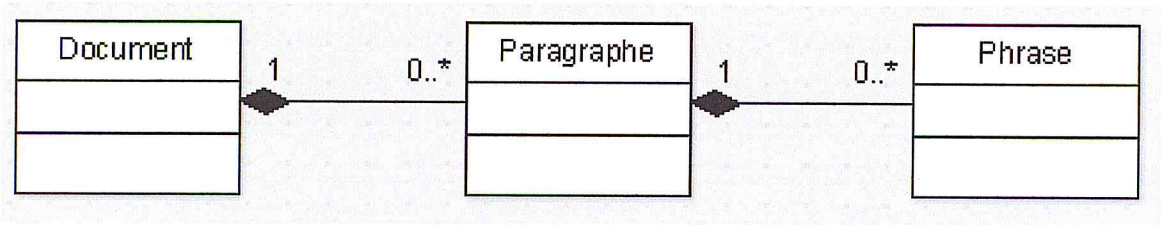


Figure 04 : *représentation graphique d'une Composition*

2.3.2. Diagramme d'activité

2.3.2.1. Definition

Un diagramme d'activités est un groupe d'activités.

Un groupe d'activités est une activité regroupant des nœuds et des arcs.

Les nœuds et les arcs peuvent appartenir à plus d'un groupe **[10], [11]**.

2.3.2.2. Activité

Une activité définit un comportement décrit par un séquencement organisé d'unités dont les éléments simples sont les actions.

Le flot d'exécution est modélisé par des nœuds reliés par des arcs (transitions).

Le flot de contrôle reste dans l'activité jusqu'à ce que les traitements soient terminés.

Une activité est un comportement (*behavior* en anglais) et à ce titre peut être associée à des paramètres **[10], [11]**.

Les Nœuds d'activité

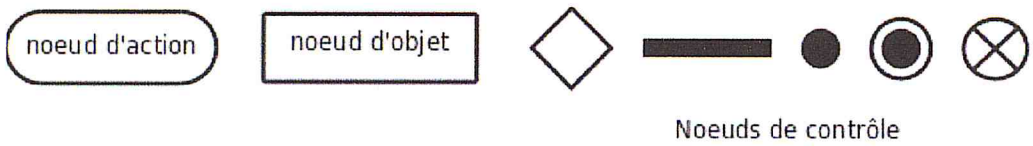


Figure 05: *Représentation graphique des nœuds d'activité*

De la gauche vers la droite, on trouve : le nœud représentant une action, qui est une variété de nœud exécutable, un nœud objet, un nœud de décision ou de fusion, un nœud de bifurcation ou d'union, un nœud initial, un nœud final et un nœud final de flot **[10], [11]**.

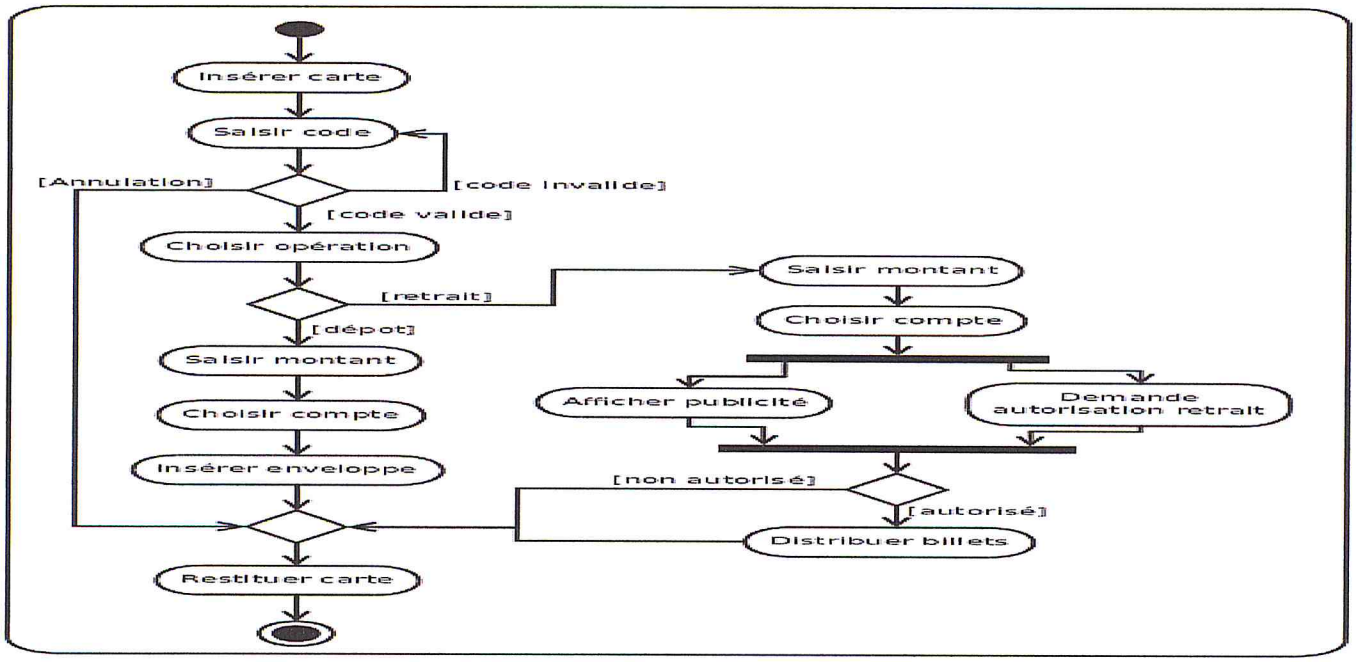


Figure 06: Exemple *d'un diagramme d'activité*

2.3.3. Le diagramme des cas (vue fonctionnelle)

2.3.4. Les cas d'utilisation

Un cas d'utilisation (use case) modélise une interaction entre le système informatique à développer et un utilisateur ou acteur interagissant avec le système.

Plus précisément, un cas d'utilisation décrit une séquence d'actions réalisées par le système qui produit un résultat observable pour un acteur.

Il y a en général deux types de description des use cases :

- une description textuelle de chaque cas,
- le diagramme des cas, constituant une synthèse de l'ensemble des cas,

Il n'existe pas de norme établie pour la description textuelle des cas.
On y trouve généralement pour chaque cas son nom, un bref résumé de son déroulement, le contexte dans lequel il s'applique, les acteurs qu'il met en jeu, puis une description détaillée, faisant apparaître le déroulement nominal de toutes les interactions, les cas nécessitant des traitements d'exceptions, les effets du déroulement sur l'ensemble du système, etc **[10], [11]**.

Les Patrons de Conception

3. Les Patrons de Conception

3.1. Patrons de Conception

3.1.1. Définition de Patrons de Conception

Les patrons (motifs ou modèles) de conception sont principalement prévus pour offrir des solutions génériques aux problèmes qui se répètent dans la conception du logiciel **[15]**, **[16]** .

Les patrons de conception, classés en 3 catégories **[28]** :

- créationnels : pour construire des composants,
- structurels : pour connecter des composants, et
- organisationnels (comportementaux) ; pour la communication entre les composants.

BUT			
PORTEE	CREATIONEL	STRUCTUREL	COMPORTEMENTAL
CLASSE	Factory Method	Adapter (class)	Interpreter
			Template Method
OBJET	Abstract Factory	Adapter (object)	Command.
	Builder	Bridge	Iterator
	Prototype	Composite	Mediator
	Singleton	Decorator	Memento
		Facade	Observer
		Flyweight	State
		Proxy	Strategy

			Visitor
			Chain Of Resp.

Tableau 01:

Les plus importants avantages des patrons sont:

- **l'abstraction** : le patron fournit une solution claire et bien fondée à un niveau élevé d'abstraction;
- **la communication** : les patrons fournissent des noms communs pour des problèmes et des solutions typiques, qui facilitent la communication parmi les développeurs de logiciel ;
- **la documentation** : un modèle décrit clairement un problème bien défini et une solution générale, dans un format fixe mais extensible ;

3.1.2. Patrons Créationnels

Les patrons créationnels concernent le processus de création d'objets. Ils créent les objets pour les clients, au lieu que les clients instancient les objets directement. Cela donne plus de flexibilité au logiciel pour décider quels objets doivent être créés dans un cas donné.

- **Fabrique Abstraite**: Fournit une interface pour créer des familles d'objet liés ou dépendants sans avoir à spécifier leurs classes concrètes.

- **Constructeur**: Sépare la construction d'un objet complexe de sa représentation, afin que le même processus de construction puisse créer différentes représentations.
- **Fabrication**: Définit une interface pour créer un objet, mais laisse aux sous-classes le soin de décider quelle classe instancier. Le pattern Fabrication laisse une classe déléguer son instanciation à des sous-classes.
- **Prototype**: Spécifie quelles sortes d'objets créer en utilisant une instance prototype, et crée de nouveaux objets en copiant ce prototype.
- **Singleton**: Fait en sorte qu'une classe n'ait qu'une seule instance, et fournit un point d'accès global à celle-ci.

3.1.3. Patrons Structuraux

Les patterns structuraux concernent la composition de classes et d'objet. Ils aident à composer des groupes d'objets en structures plus larges, comme par exemple des interfaces utilisateur complexes ou des données comptables.

- **Adaptateur**: Convertit l'interface d'une classe en une autre interface attendue par les clients. L'adaptateur permet de travailler ensemble à des classes pour lesquelles ce serait normalement impossible à cause d'interfaces incompatibles.
- **Pont**: Sépare une abstraction de son implémentation de sorte que les deux puissent varier indépendamment.
- **Composite**: Compose des objets en structures d'arbres pour représenter des hiérarchies. Composite permet à des client de traiter des objets ou des composition d'objets de la même manière.
- **Décorateur**: Attache des responsabilités additionnelles à un objet de manière dynamique. Les Décorateurs fournissent une alternative flexible à l'héritage (sous-classement) pour étendre les fonctionnalités.

- **Façade**: Fournit une interface unifiée à un ensemble d'interfaces dans un sous-système. La Façade définit une interface de plus haut niveau qui rend le sous-système plus facile à utiliser.
- **Poids mouche**: Utilise un procédé de partage pour supporter de petits objets efficacement.
- **Proxy**: Fournit un substitut à un objet pour contrôler l'accès à celui-ci.

3.2. Patrons Comportementaux

Les patterns comportementaux caractérisent les façons dont les classes et les objets interagissent et se partage les responsabilités. Ils aident à définir la communication entre les objets du système et comment le flot d'information est contrôlé dans un programme complexe.

- **Chaîne de responsabilité**: Evite de coupler l'émetteur d'une requête à son récepteur en donnant à plus d'un objet la chance de traiter la requête. Les objets récepteurs sont chaînés et passent la requête le long de la chaîne jusqu'à ce qu'un objet la traite.
- **Commande**: Encapsule une requête en tant qu'objet, permettant ainsi de paramétrer les clients avec des requêtes différentes, de mettre les requêtes en file d'attente, et de supporter l'annulation des opérations.
- **Interpréteur**: Pour un langage donné, définit une représentation de sa grammaire avec un interpréteur qui utilise la représentation pour interpréter les phrases du langage.
- **Itérateur**: Fournit un moyen d'accéder séquentiellement aux éléments d'un objet agrégé sans exposer sa représentation sous-jacente.
- **Médiateur**: Définit un objet qui encapsule la façon d'interagir d'un groupe d'objets. Le Médiateur favorise un couplage lâche en

empêchant les objets de faire référence aux autres explicitement, et permet de faire varier leurs interactions indépendamment.

- **Mémento**: Sans violer le principe d'encapsulation, capture et externalise l'état interne d'un objet, de sorte que cet objet puisse être restauré dans cet état plus tard.
- **Observateur**: Définit une relation un à plusieurs entre des objets de sorte que lorsqu'un objet change d'état, tous les objets dépendants sont notifiés et mis à jour automatiquement.
- **Etat**: Permet à un objet de modifier son comportement lorsque son état change. On aura l'impression que l'objet change de classe.
- **Stratégie**: Définit une famille d'algorithmes, encapsule chacun, et les rend interchangeables. La Stratégie permet à un algorithme de varier indépendamment des clients qui l'utilisent. Strategy lets the algorithm vary independently from the clients that use it.
- **Patron de méthode**: Définit le squelette d'un algorithme dans une opération, en reportant certaines étapes à des sous-classes. Le Patron de méthode permet à des sous-classes de redéfinir certaines étapes d'un algorithme sans changer la structure de l'algorithme.
- **Visiteur**: Représente une opération à accomplir sur les éléments de la structure d'un objet. Le Visiteur permet de définir une nouvelle opération sans changer les classes des éléments sur lesquels il opère.

3.3. Patrons Architecturaux

Un modèle architectural adresse le principe pour la structuration globale de l'architecture du logiciel.

Shaw a identifié sept modèles qui guident la conception de système à niveau élevé et ont discuté la manière qu'ils guident la composition des systèmes des types particuliers de composants **[27]**.

Buschman et al **[29]** présentent le patron *Model-View-Controller* comme un patron architecturel.

Les Graphes

4. Les Graphes

4.1. Présentation

Les graphes représentent un instrument puissant pour modéliser de nombreux problèmes combinatoires, qui seraient sans cela difficilement abordables par des techniques classiques **[19]**.

4.2. Définition de graphe [17]

Définition 1

Un graphe orienté $G=(X, U)$ est déterminé par les données suivantes :

- Un ensemble X dont les éléments sont appelés sommets. Si $N=card(X)$ est le nombre de sommets, on dit que le graphe G est d'ordre N.

- Un ensemble U dont les éléments sont des couples ordonnés de sommets appelés arcs. Si $u= (i, j)$ est un arc de G. i est l'extrémité initiale de u et j est l'extrémité terminale de u. on notera $card\ (U)=M$.

Définition 2

Un multi graphe est un couple $G=(X, U)$, dans lequel X est un ensemble de sommets, et U est une famille d'arcs $U= (u_1, u_2, \ldots, u_m)$ [43]. Cette définition permet de traiter des graphes dont plusieurs arcs auraient la même origine et la même extrémité, d'où le nom de multi graphe.

Définition 3

Un graphe orienté valué $G=(X, U, W)$. X désigne un ensemble de N sommets et U un ensemble de M arcs. $W (i, j)$, aussi noté W_{ij}, est l'évaluation (aussi appelée poids ou coût) de l'arc (i, j), par exemple une distance, un coût de transport, ou un temps de parcours.

4.3. Correspondance de Graphe

Les graphes constituent un mode de représentation fréquemment utilise dans le domaine des sciences et technologies de l'information qui permettent a la description de données structurées.
Un graphe G est un ensemble V de nœuds et un ensemble E d'arcs, G = (V, E).
Les outils de classification supervisée sont de plus en plus nécessaires dans de nombreuses applications telles que la reconnaissance des formes , la CBR (*Case Based Reasoning*) , l'analyse des composantes chimiques .
Dans le cas du problème de reconnaissance des formes, étant donne deux graphes : le graphe de modèle GM et le graphe de données GD, la procédure de comparaison implique de vérifier si ils sont similaires ou non.

De manière générale, nous pouvons représenter le problème de la correspondance de graphe comme suit : Etant donne deux graphes GM = (VM, EM) et GD = (VD, ED), avec | VM | = | VD |, le problème est de trouver une fonction de correspondance f: VD → VM, tel que (u, v) Î ED Si et seulement si $(f(u), f(v))$ Î EM.

Lorsqu'une telle fonction de correspondance *f* existe, nous somme en présente d'un isomorphisme, et GD est dit d'être isomorphe a GM et ce type s'appelle □ correspondance exacte □.
D'autre part, le terme □ inexact □ appliquée aux problèmes de la correspondance de graphe, indique qu'il n'est pas possible de trouver un isomorphisme entre les deux graphiques.
C'est le cas lorsque le nombre de sommets ou le nombre d'arcs sont différents a la fois dans le graphe modèle et graphe de données. Dans ce cas la, on peut trouver la meilleure correspondance entre eux en trouvant une correspondance non-bijective entre le graphe de données et le graphe de modèle.
Le problème de la correspondance de graphe a été prouve être le NP-complet.
Lorsque le nombre de nœuds dans les deux graphes sont différents, le problème de la correspondance de graphe devient plus difficile que dans le cas de la correspondance de graphe exact.
De même, la complexité du problème de sous-graphe inexact est équivalente a la complexité du problème de la plus grand sous graphe commun, qui est aussi connu pour être NP-complet. Plusieurs techniques ont été proposées pour résoudre ce problème, par exemple, la relaxation probabiliste, l'algorithme EM les réseaux de neurones , des arbres de décision et un algorithme génétique .
Toutes les méthodes énoncées antérieurement ont comme point commun l'utilisation d'un algorithme d'optimisation pour adapter un graphe.

Pour mesure la bonne similarité entre deux graphes.
Cette fonction est conçue en tenant compte du cout pour faire la correspondance $VD \rightarrow VM$.
Les auteurs sont convaincus qu'une correspondance convenable doit conduire a une distance entre graphe précise.
Selon cette hypothèse, le problème est tourne en une question de distance entre graphes.
De plus, ce point de vue sur le problème de la correspondance de graphe permettra de lancer un banc de tests sur notre approche et de fournir une étude comparative.

CHAPITRE 3:

Ingénierie Dirigée Par Les Modèles

Chapitre 03:

Ingénierie Dirigée Par Les Modèles

1. Ingénierie Dirigée Par Les Modèles

1.1. Introduction

En novembre *2000*, l'Object Management Group *(OMG)* [18] a proposé une approche nommée Model Driven Architecture *(MDA™)* pour le développement et la maintenance des systèmes à prépondérance logicielle .

En *2003*, le CNRS crée l'AS « MDA ». Après deux ans de veille technologique et de travaux de recherche, les membres de l'AS défendent alors l'idée que ce vaste mouvement mondial initié par l'industrie est une révolution culturelle dans le contexte du développement de logiciels.

De plus, la portée de ce mouvement ne doit pas se limiter à des avancées technologiques guidées par l'**OMG**, mais au contraire s'élargir pour tenter d'établir une synergie entre les travaux de recherche présents et passés manipulant eux aussi des modèles.
Pour cette raison, nous ne parlerons plus de **MDA** dans ce document mais d'*Ingénierie Dirigée par les Modèles* (**IDM** en Français ou **MDE** pour "Model Driven Engineering").

Nous affirmons ainsi que cette évolution rapide et importante des pratiques industrielles de production de logiciels pour être durable doit s'appuyer sur la convergence des travaux issus de différents domaines de l'ingénierie logicielle. Dans ce cadre, la recherche doit jouer un rôle très important en établissant des *ponts interdisciplinaires* pour profiter au mieux des avancées technologiques de chacun en proposant un *cadre intégrateur solide*.

Pour faire face à la complexité et à l'évolution croissante des applications, l'**IDM** ouvre de nouvelles voies d'investigation. En autorisant une appréhension des applications selon différents points de vues tout en intégrant comme fondamental la composition et mise en cohérence de ces perspectives, elle ne peut s'inscrire dans la pérennité que si elle prend ces racines dans des bases bien fondées établies par la théorie **[31]**.

1.2. L'Ingénierie dirigée par les modèles

L'Ingénierie Dirigée par les Modèles (**IDM**) est une discipline qui a pour vocation l'automatisation et la sûreté du développement des systèmes logiciels complexes notamment les systèmes embarqués en fournissant des outils et des langages permettant la transformation de modèles d'un niveau d'abstraction à un autre ou d'un espace technologique à un autre.

L'IDM est le domaine de recherche en pleine émergence qui considère les modèles comme les éléments de base dans la production, le fonctionnement et l'évolution des systèmes d'information, et mettant à disposition des outils, concepts et langages pour créer et transformer des modèles.
Ce que propose l'approche de l'ingénierie des modèles (IDM) est simplement de mécaniser le processus que les ingénieurs expérimentés suivent à la main.
L''IDM à des plusieurs aspects modélisation, transformation et applications industrielles. **[26], [25]**.

1.3. Les Modèles

L'Ingénierie Dirigée par les Modèles se focalise sur des modèles et comment ceux-ci peuvent être utilisés pour créer un système.
La structure d'un modèle est définie par une syntaxe abstraite du langage dans lequel est implémenté ce modèle.
La présentation d'un modèle est définie par la syntaxe concrète du langage de modélisation.
Dans la ***section 4.3*** nous présenterons de manière plus précise la notion de structure.

Les modèles offrent de nombreux avantages dont le plus important est de spécifier différents niveaux d'abstraction, facilitant la gestion de la complexité inhérente aux applications.
Les modèles possédant un niveau élevé d'abstraction sont utilisés pour présenter l'architecture générale d'une application ou sa place dans une organisation, tandis que les modèles très concrets permettent de spécifier précisément des protocoles de communication réseau ou des algorithmes de synchronisation.
Même si les modèles se situent à des niveaux d'abstraction différents, il est possible d'exprimer des relations de raffinement entre eux **[32] [31]**.
De véritables liens de traçabilité et des relations garantissent la cohérence d'un ensemble de modèles impliqués dans une même application.
L'objectif majeur de l'**IDM** est l'élaboration de modèles pérennes, indépendant des détails techniques des plateformes d'exécution **(J2EE, .Net, PHP, etc.),** afin de permettre la génération automatique du code des applications et d'obtenir un gain significatif de productivité.
Le principe clé de l'**IDM** consiste en l'utilisation de modèles aux différentes phases du cycle de développement d'une application. Plus précisément, l'**IDM** préconise l'élaboration de modèles d'exigence (**CIM**), d'analyse et de conception (**PIM**) et de code (**PSM**).
Nous pouvons donc catégoriser les modèles de la manière suivante :

- Les modèles dits **CIM** (Computation Independent Model).
- Les modèles dits **PIM** (Platform Independent Model).
- Les modèles dits **PSM** (Platform Specific Model).

Dans les sections suivantes nous présenterons l'ensemble de ces types de modèles **[21], [22]**.

1.3.1. Modèle CIM

C'est le modèle métier ou le modèle du domaine d'application.
Le **CIM** permet la vision du système dans l'environnement où il opèrera, mais sans rentrer dans le détail de la structure du système ni de son implémentation. Il aide à représenter ce que le système devra exactement faire. Il est utile, non seulement comme aide pour comprendre un problème, mais également comme source de vocabulaire partagé par d'autres modèles.
L'indépendance technique de ce modèle lui permet de garder tout son intérêt au cours du temps et il est modifié uniquement si les connaissances ou les besoins métier changent.
Le savoir-faire est recentré sur la spécification **CIM** au lieu de la technologie d'implémentation. Dans les constructions des **PIM** et des **PSM**, il est possible de suivre les exigences modélisées du **CIM** qui décrivent la situation dans lequel le système est utilisé, et réciproquement **[22] ,[23]**.

1.3.2. Modèle PIM

Un modèle **PIM** présente une indépendance vis-à-vis de toute plateforme technique **(EJB, CORBA, .NET, etc.)** et ne contient pas d'informations sur les technologies qui seront utilisées pour déployer l'application.
C'est un modèle informatique qui représente une vue partielle d'un **CIM**.
Le **PIM** représente la logique métier spécifique au système ou le modèle de conception.
Il représente le fonctionnement des entités et des services.
Il doit être pérenne au cours du temps.
Il décrit le système, mais ne montre pas les détails de son utilisation sur la plateforme.
A ce niveau, le formalisme utilisé pour exprimer un **PIM** est un diagramme de classes en UML qui peut être couplé avec un langage de contrainte comme OCL (Object Constraint Language) .
Il existe plusieurs niveaux de **PIM**.
Le **PIM** peut contenir des informations sur la persistance, les transactions, la sécurité, etc.
Ces concepts permettent de transformer plus précisément le modèle PIM vers le modèle **PSM [21], [22]**.

1.3.3. Modèle PSM

Un modèle **PSM** est dépendant de la plateforme technique spécifiée par les architectes d'un système.

Le **PSM** sert essentiellement de base à la génération de code exécutable vers la ou les plateformes techniques.
Le **PSM** décrit comment le système utilisera cette ou ces plateformes. Il existe plusieurs niveaux de **PSM**.
Le premier, issu de la transformation d'un **PIM**, se représente par un schéma **UML** spécifique à une plateforme.
Les autres **PSM** sont obtenus par transformations successives jusqu'à l'obtention du code dans un langage spécifique **(Java, C++, C#, etc.)** Un PSM d'implémentation contiendra par exemple des informations comme le code du programme, les types pour l'implémentation, les programmes liés, les descripteurs de déploiement **[22],[21]**.

1.4. Transformation de modèles

La définition la plus générale et qui fait l'unanimité au sein de la communauté **[23] [09]** IDM consiste à dire qu'*une transformation de modèles est la génération d'un ou de plusieurs modèles cibles à partir d'un ou de plusieurs modèles sources*.

Dans l'approche par modélisation, cette transformation se fait par l'intermédiaire de règles de transformations qui décrivent la correspondance entre les entités du modèle source et celles du modèle cible.

En réalité, la transformation se situe entre les métamodèles source et cible qui décrivent la structure des modèles cible et source.

Le moteur de transformation de modèles prend en entrée un ou plusieurs modèles sources et crée en sortie un ou plusieurs modèles cibles.

Une transformation des entités du modèle source met en jeu deux étapes :

1. ***La première étape*** : permet d'identifier les correspondances entre les concepts des modèles source et cible au niveau de leurs métamodèles, ce qui induit l'existence *d'une fonction de transformation* applicable à toutes les instances du métamodèle source.

2. ***La seconde étape*** : consiste à appliquer la transformation du modèle source afin de générer automatiquement le modèle cible par un programme appelé *moteur de transformation* ou d'exécution.

L'approche par programmation, l'approche par Template et l'approche par modélisation.

L'approche par programmation consiste à utiliser les langages de programmation en général, et plus particulièrement les langages orientés objet. Dans cette approche, la transformation est décrite sous forme d'un programme informatique à l'image de n'importe quelle application informatique. Cette approche reste très utilisée car elle réutilise l'expérience accumulée et l'outillage des langages existants.

L'approche par Template consiste à définir des canevas des modèles cibles souhaités. Ces canevas sont des *modèles cibles paramétrés* ou des *modèles Template*.

L'exécution d'une transformation consiste à prendre un modèle Template et à remplacer ses paramètres par les valeurs d'un modèle source.

L'approche par modélisation consiste quant à elle à appliquer les concepts de L'ingénierie des modèles aux transformations des modèles elles-mêmes. L'objectif est de modéliser les transformations de modèles et de rendre les modèles de transformation pérennes et productifs, en exprimant leur indépendance vis-à-vis des plates-formes d'exécution.

Le standard **MOF 2.0** QVT de l'**OMG** a été élaboré dans ce cadre et a pour but de définir un métamodèle permettant l'élaboration des modèles de transformation de modèles.
A titre d'exemple, cette approche a été choisie par le ***groupe ATLAS*** à travers son langage de transformation de modèles ***ATL*** que nous verrons dans la section suivante **[13] [14] [18]**.

1.4.1. Types de transformations

Une transformation de modèles met en correspondance des éléments des Modèles cibles et sources.
Dans **[33]** on distingue les types de transformation suivants :

On distingue les types de transformation suivants :

- une transformation simple (***1 vers 1***) qui associe à tout élément du modèle Source au plus un élément du modèle cible. Un exemple typique de cette situation est la transformation d'une classe UML munie de ses opérations et de ses attributs en une classe homonyme en Java ;
- une transformation multiple (***M vers N***) prend en entrée un ensemble d'éléments du modèle source et produit un ensemble d'éléments du modèle cible.

Les transformations de décomposition de modèles (*1 vers N*) et de fusion de modèles (*N vers 1*) sont des cas particuliers de transformations multiples ;

- une transformation de mise à jour encore appelée transformation sur place consiste à modifier un modèle par ajout, modification ou suppression d'une partie de ses éléments.
  Dans ce type de transformation, les modèles source et cible sont confondus. Une telle transformation agit directement sur le modèle source sans créer de modèle cible.
  Un exemple typique d'une telle transformation est la restructuration de modèles (***Model Refactoring***) qui consiste à réorganiser les éléments du modèle source afin d'en améliorer sa structure ou sa lisibilité .

1.4.2. Axes de transformation

La transformation de modèles peut se faire selon trois axes de transformation **[33] :**

- ***l'axe processus*** permet de positionner fonctionnellement les transformations par rapport au processus global d'ingénierie. Il est composé de deux sous-axes : le sous-axe *vertical* qui consiste à faire une transformation de modèles en changeant de niveau d'abstraction (*par exemple* ***: raffinement d'un modèle, passage de PIM vers PSM***), et le sous-axe *horizontal* qui consiste à faire une transformation de modèles en restant au même niveau d'abstraction (*par exemple* ***: Restructuration de modèle ou Model Refactoring***) ,

- ***l'axe métamodèle*** permet de caractériser l'importance des métamodèles mis en jeu dans une transformation. Par analogie avec la programmation classique, un algorithme vérifie la conformité des types des variables mis en jeu dans une opération.
  Nous pouvons faire le parallélisme avec un algorithme de transformation de modèles qui permet de faire des opérations entre modèles (différence, Composition, fusion, etc.).
  Dans le cas de la transformation, les métaclasses des métamodèles correspondent aux types des variables, d'où l'influence des métamodèles dans le code d'une transformation de modèles.
  Exemple de transformation utilisant l'axe métamodéle***: UML to SysML*** (SysML-website) ,

- ***l'axe paramétrage*** permet de caractériser le degré d'automatisation des

Transformations avec la présence de données pour paramétrer la transformation.
Le passage des informations à une transformation de modèles peut être soit interne (par exemple, des valeurs ou des relations fixées dans des règles), soit transmis à la transformation (par exemple, le modèle source). Les informations transmises à la transformation sont appelées paramètres de la transformation.
Nous parlerons de transformation *automatique* si tous les paramètres sont mis à la disposition de la transformation avant son exécution et de transformation ***semi-automatique*** si certains paramètres ne sont renseignés qu'à l'exécution de la transformation .

1.4.3. Taxonomie des transformations

Partant de la nature des métamodèles source et cible, on distingue encore selon la classification adoptée aux transformations dites ***endogènes*** et ***exogènes*** combinées à des transformations dites **verticales** et ***horizontales [33]***.

Une transformation est dite ***endogène*** si les modèles cible et source sont issus du même métamodèle, et ***exogène*** dans le cas contraire.

Une transformation simple ou multiple peut être ***exogène*** ou ***endogène*** selon la nature des métamodèles source et cible impliqués dans la transformation.
Par contre une transformation sur place implique un même métamodèle, donc elle est ***endogène***.

Une démarche de transformation peut aussi induire un changement de niveau d'abstraction.

Une transformation est dite ***verticale*** si elle met en jeu différents niveaux d'abstraction dans la transformation.
Le passage de **PIM** vers **PSM** ou rétro conception est une transformation ***exogène et verticale*** alors que le raffinement est une transformation **endogène** et **verticale**.

Une transformation est dite ***horizontale*** lorsque les modèles source et cible impliqués dans la transformation sont au même niveau d'abstraction.

La restructuration, la normalisation et l'intégration des patrons sont des exemples de transformation ***endogène*** et ***horizontale*** , tandis que la migration des plates-formes et la fusion de modèles sont des exemples de transformation ***exogène et horizontale***.

Il est important de noter que les modèles source et cible peuvent appartenir à des espaces technologiques différents.
Ci-dessous résume les combinaisons possibles entre transformations de modèles.

FUJABA

FUJABA [34], acronym de *From UML to Java and Back Again*, a pour but de fournir un environnement de génération de code Java et de rétro-conception.
Il utilise UML comme langage de modélisation visuel.
Durant la dernière décennie, l'environnement de **FUJABA** est devenu une base pour plusieurs activités de recherche notamment dans le domaine des applications distribuées, ses systèmes de bases de données ainsi que dans le domaine de la modélisation et de la simulation des systèmes mécaniques et électriques. Ainsi, l'environnement de **FUJABA** est devenu un projet open-source qui intègre les mécanismes de l'IDM.

1.5. Les langages/outils dédiés à la transformation de modèles

Dans cette catégorie, on retrouve les outils et langages conçus spécifiquement pour faire de la transformation de modèles et prévus pour être plus ou moins intégrables dans les environnements de développement standard. Parmi ces outils nous pouvons citer Mia-Transformation de Mia-Software, et le plug-in ADT qui implémente le langage **ATL** du groupe **ATLAS** de ***l'INRIA-LINA***. Dans la suite de cette section, nous présentons essentiellement le langage **ATL** qui est représentatif de cette catégorie des langages dédiés aux transformations, et fait l'objet de nombreuses expérimentations dans la communauté IDM.

ATL

ATL [13][14], acronyme de ***ATLAS Transformation Language*** , est un langage à vocation déclarative, mais en réalité hybride, qui permet de faire des transformations de modèles aussi bien endogènes qu'exogènes.
Les outils de transformation liés à **ATL** sont intégrés sous forme de plug-in ***ADT (ATL Development Tool)*** pour l'environnement de développement Eclipse.
Un modèle de transformation **ATL** se base sur des définitions de métamodèles au format XMI.

Sachant qu'il existe des dialectes d'XMI, **ADT** est adapté pour interpréter des métamodèles décrits à l'aide d'EMF (Eclipse) ou MDR (NetBeans).
Afin d'assurer son indépendance par rapport aux autres outils de modélisation, **ADT** met à disposition le langage KM3 (Kernel MetaMetaModel) qui est une forme textuelle simplifiée d'EMOF permettant de décrire des métamodèles et des modèles.
En effet, **ADT** supporte les modèles décrits dans différentes dialectes d'XMI, qui peuvent par exemple être décrits au format KM3 et transformés au format XMI voulu.
Développement logiciel orienté modèles

ATL est défini par un modèle MOF pour sa syntaxe abstraite et possède une Syntaxe concrète textuelle.
Pour accéder aux éléments d'un modèle, **ATL** utilise des requêtes sous forme d'expressions OCL.
Une requête permet de naviguer entre les éléments d'un modèle et d'appeler des opérations sur ceux-ci.
Une règle déclarative d'**ATL**, appelée M*atched Rule*, est spécifiée par un nom, un ensemble de patrons sources *(InPattern) mappés* avec les éléments sources, et un ensemble de patrons cibles (*OutPattern*) représentant les éléments créés dans le modèle cible.
Depuis la version 2006 d'ATL, de nouvelles fonctionnalités ont été ajoutées telles que l'héritage entre les règles et le multiple *pattern matching* (plusieurs modèles en entrée).
Le style impératif d'ATL est supporté par deux constructions différentes.
En effet, on peut utiliser soit des règles impératives appelées C*alled Rule*, soit un bloc d'instructions impératives (*ActionBlock*) utilisé avec les deux types de règles.
Une *Called Rule* est appelée explicitement en utilisant son nom et en initialisant ses paramètres **[14].**

1.5.1.Outils de métamodélisation

La dernière catégorie des outils de transformation de modèles est celle des outils de méta-modélisation dans lesquels la transformation de modèles revient à L'exécution d'un métaprogramme.
Parmi ces outils, nous pouvons citer **Kermeta [36]** de *l'IRISA-INRIA Rennes, XMF-Mosaic* **[35]** de la société *Xactium*, **EMF/Ecore [38]** de la fondation Eclipse, et l'outil **TOPCASED [33].**

KERMETA

Dans l'approche par programmation classique, on dit qu'un programme est composé d'un ensemble de structures de données combinées à des algorithmes.
C'est sur la base de cette assertion que le langage de metamodèlisation **KERMETA [36],[37]** a été élaboré. une description en **KERMETA** est assimilable à un programme issu de la fusion d'un ensemble de métadonnées (EMOF) et du métamodèle d'action AS (Action Semantics) qui est maintenant intégré dans UML 2.0 Superstructure.
Le langage KERMETA est donc une sorte de dénominateur commun des langages qui coexistent actuellement dans le paysage de l'IDM.
Ces langages sont les langages de métadonnées (EMOF, ECORE,…), de transformation de modèles (QVT, ATL,…), de contraintes et de requêtes (OCL), et d'action (Action Semantics, Xion).
Cette composition fait de **KERMETA** un véritable langage de métamodélisation exécutable.
Comme UML 2.0 Infrastructure, **KERMETA** intervient à deux niveaux dans l'architecture de métamodélisation de l'OMG.
D'une part il intervient comme un langage de niveau M3 c'est-à-dire que tous les métamodèles lui sont conformes, mais également comme une bibliothèque de base pour construire des métamodèles de niveau M2.

EMF/ECORE

EMF [38] qui signifie *Eclipse Modeling Framework*, est une plate-forme de modélisation et de génération de code qui facilite la construction d'outils et d'autres applications basées sur des modèles structurés.
Il permet le développement rapide et l'intégration de nouveaux plug-ins Eclipse.
EMF est composé d'un ensemble de briques appelées plug-ins , parmi ces plug-ins nous pouvons citer :

- le métamodèle *Ecore* qui est un canevas de classes pour décrire les modèles,
- *EMF* et manipuler les référentiels de modèles,
- *EMF.Edit* qui est un canevas de classes pour le développement d'éditeurs de modèles EMF ,
- le modèle de génération *GenModel* qui permet de personnaliser la génération Java,
- *JavaEmitterTemplate* qui est un moteur de template générique,
- *JavaMerge* qui est un outil de fusion de code Java.

1.6. Evolutions récentes en Ingénierie Dirigée par les Modèles

1.6.1. Le MDA et l'IDM

L'idée initiale de l'**OMG** consistait à s'appuyer sur le standard **UML** pour décrire séparément les parties des systèmes indépendantes de la plate-forme spécifique (**PIM** ou Platform Independant Models) et les parties liées aux plates-formes (**PSM** ou Platform Specific Models).
Dans les années qui ont suivi, ce projet est devenu plus ambitieux et a évolué de façon interne et de façon externe.
De façon interne d'abord, l'**OMG** met surtout en avant actuellement une architecture dont le socle est le **MOF** (Meta-Object Facility) **[7]**.
Sur ce socle s'appuient d'une part une collection de métamodèles dont UML n'est qu'un élément parmi d'autres, et d'autre part une technologie émergente de transformation de modèles nommée **QVT [31].**
De façon externe ensuite, l'approche **MDA** devient une variante particulière de l'Ingénierie Dirigée par les Modèles.

L'**IDM** peut être vue comme une famille d'approches qui se développent à la fois dans les laboratoires de recherche et chez les industriels impliqués dans les grands projets de développement logiciels.
Deux de ces industriels (**IBM et Microsoft**) ont récemment défini leur stratégie **MDE** et le moins que l'on puisse dire c'est qu'elles semblent converger.

1.6.2. UML et l'IDM

Il faut donc séparer clairement les approches **IDM** du formalisme **UML**.
Non seulement la portée de l'**IDM** est beaucoup plus large que celle d'**UML**, mais la vision **IDM** est aussi très différente de celle d'**UML** et parfois même en contradiction.
UML est, dans ses versions 1.5 et 2.0, un standard assez monolithique obtenu par consensus à maxima, dont on doit réduire la portée à l'aide de mécanismes comme les profils.
Ces mécanismes n'ont pas toute la précision souhaitable et mènent parfois à des contorsions dangereuses pour se rapprocher d'**UML**.
Dans certains outils **UML** de première génération, le support des profils **UML** a été présenté comme une fonctionnalité importante.
De plus certains de ces outils ont proposé des langages de décoration propriétaires.

Le résultat de ces choix est bien souvent de créer des modèles "partimoniaux" (legacy model) qui ont parfois coûté cher à produire et qui vont être difficiles à réintégrer dans des approches **IDM [31]**.
Ces modèles patrimoniaux restent liés à l'outil qui les a produits, ce qui est en contradiction avec les objectifs de l'approche de l'OMG prônant l'indépendance de la plate-forme et donc à fortiori des outils de développement.
Microsoft ne cache pas que son utilisation d'UML restera probablement essentiellement «contemplative » ou encore documentaire, c'est à dire utilisable essentiellement pour produire et communiquer des esquisses.
La déclinaison plus solide de l'**IDM** chez Microsoft, celle qui est progressivement intégrée dans l'outillage Visual Studio, s'appelle "Software Factories".
Elle est fondée essentiellement sur des langages de domaines ***(Domain Specific Languages ou DSL)*** **[7]** de petite taille, facilement manipulables, transformables, combinables, etc.
En un mot ces **DSL** sont la base de l'automatisation de l'**IDM** chez Microsoft.
IBM ne dit pas autre chose dans son manifeste.
Les trois axes de l'Ingénierie Dirigée par les Modèles sont d'après IBM (1) les standards ouverts, (2) l'automatisation et (3) la représentation
Parmi les standards ouverts UML peut bien sûr avoir sa place, mais ce sera aux cotés de XML et d'autres standards. Par contre, ce qui est essentiel, c'est la possibilité de traitement automatique de modèles (par exemple tissage, vérification, transformation, etc.) s'appuyant sur des standards précis, limités en taille et spécialisés.
On retrouve encore ici cette idée des **DSL**.
Les **DSL** sont ici de petits langages spécifiques de domaines adaptés à des corporations particulières ou à des besoins particuliers.
C'est ce que le manifeste IBM appelle la "représentation directe", c'est-à-dire la mise à disposition de métiers particuliers ou de tâches spécifiques de langages précis et outillés (éditeurs, générateurs, vérificateurs, etc.).
La vision de Microsoft est progressivement mise en œuvre dans Visual Studio.
La vision d'IBM est mise en œuvre par exemple dans l'outillage **EMF** (**E**clipse **M**odeling **F**ramework).

Ces visions correspondent à l'architecture multi-niveaux de l'**OMG [7]** fondée sur la pyramide de méta-modélisation dominée par le **MOF** ainsi que sur la possibilité de définir une grande variété de métamodèles spécialisés pour les **DSL**.

CHAPITRE 4:

ANALYSE ET CONCEPTION

Chapitre 04:

Analyse et Conception

1. Analyse et Conception

1.1. Introduction

Cette partie est consacrée aux étapes fondamentales pour le développement de notre application du domaine du m Model Driven Development.

Pour la conception et la réalisation de notre application, nous avons choisi de modéliser avec le formalisme UML (Unified Modeling Language) qui offre une flexibilité marquante qui s'exprime par l'utilisation des diagrammes.

1.2. Spécification des besoins

C'est une étape primordiale au début de chaque démarche de développement. Son but est de veiller à développer un logiciel adéquat, sa finalité est la description générale des fonctionnalités du système, en répondant à la question :

Quelles sont les fonctions du système ?

Notre système doit principalement réaliser les exigences suivantes :

I. La première partie est la réalisation d'une interface graphique qui permet au client de réaliser leur diagramme UML (diagramme de classe) de leur société.

II. Relier tous les classe java t'elle que a été relier dans le modèle source,
III. L'application java construite doit être compilé et exécuter de manière automatique.

1.2.1. Diagramme De Cas D'utilisation

Un cas d'utilisation est utilisé pour définir le comportement d'un système ou la sémantique de toute autre entité sans révéler sa structure interne. Chaque cas d'utilisation spécifie une séquence d'action.

La responsabilité d'un cas d'utilisation est de spécifier un ensemble d'instances, où une instance de cas d'utilisation représente une séquence d'actions que le système réalise et qui fournit un résultat observable par l'acteur.

Voici les cas d'utilisation générale de notre application :

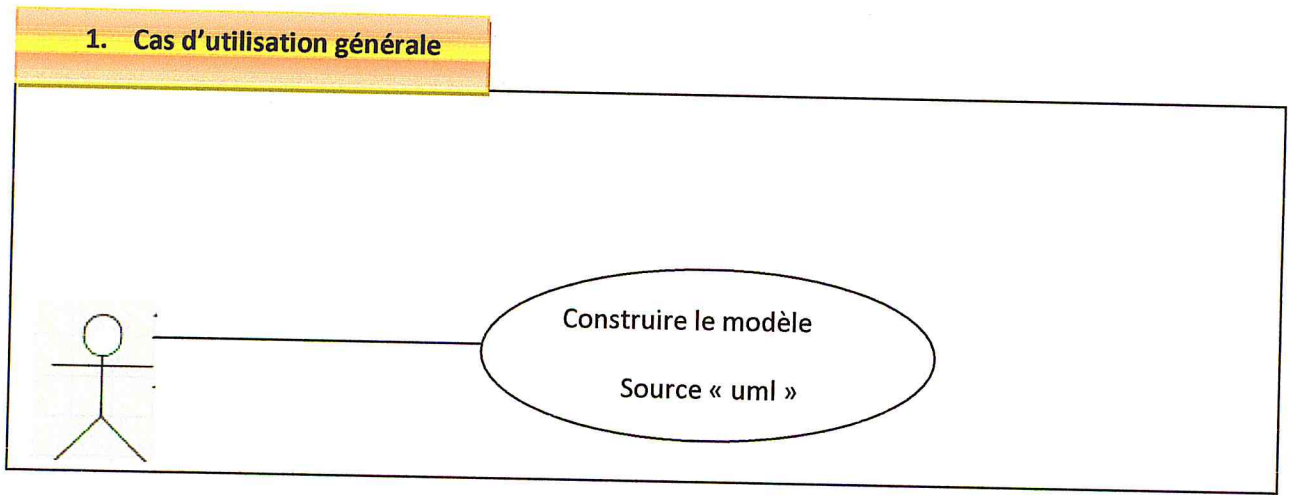


FIGURE 07: *Diagramme De Cas D'utilisation Générale*

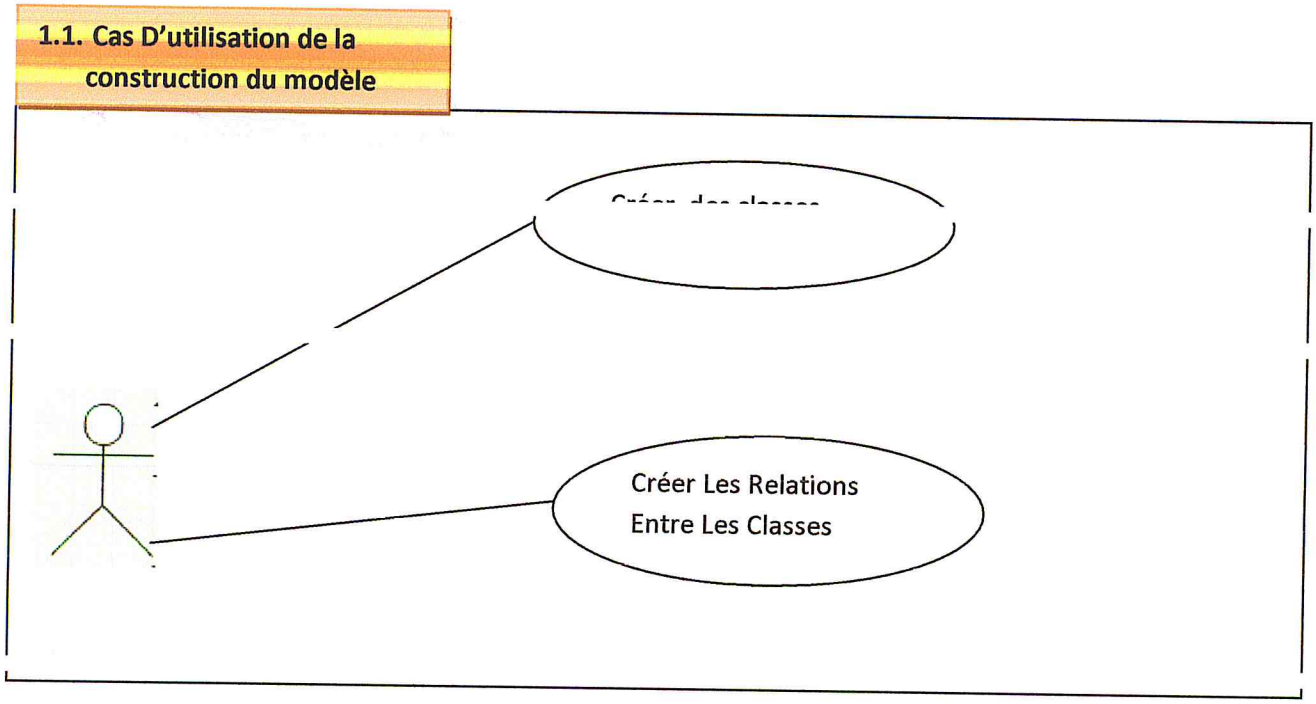


FIGURE 08: *Diagramme De Cas D'utilisation De Construction Du Modèle Source*

Cette figure représente un cas d'utilisation de la tache principale dans notre application « La construction du modèle source »

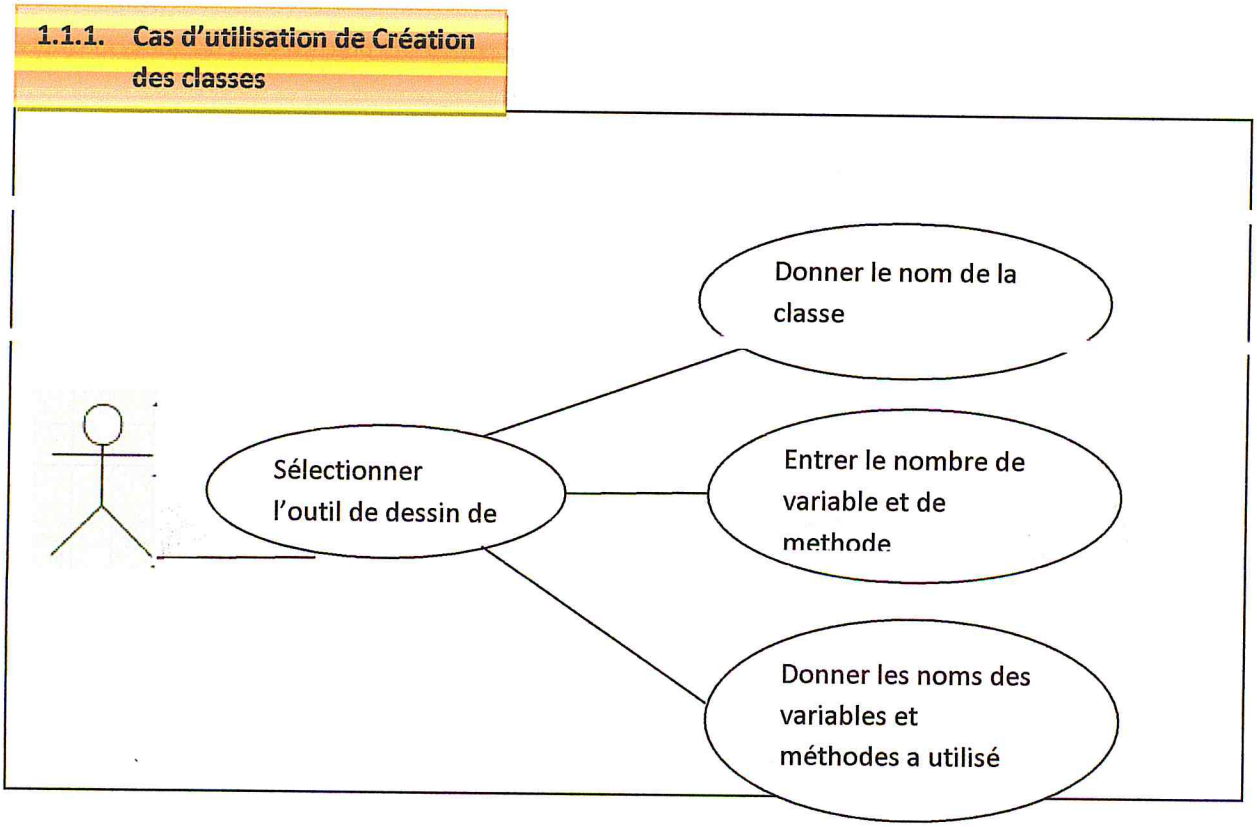


FIGURE 09 : *Diagramme De Cas D'utilisation De Création Des Classes*

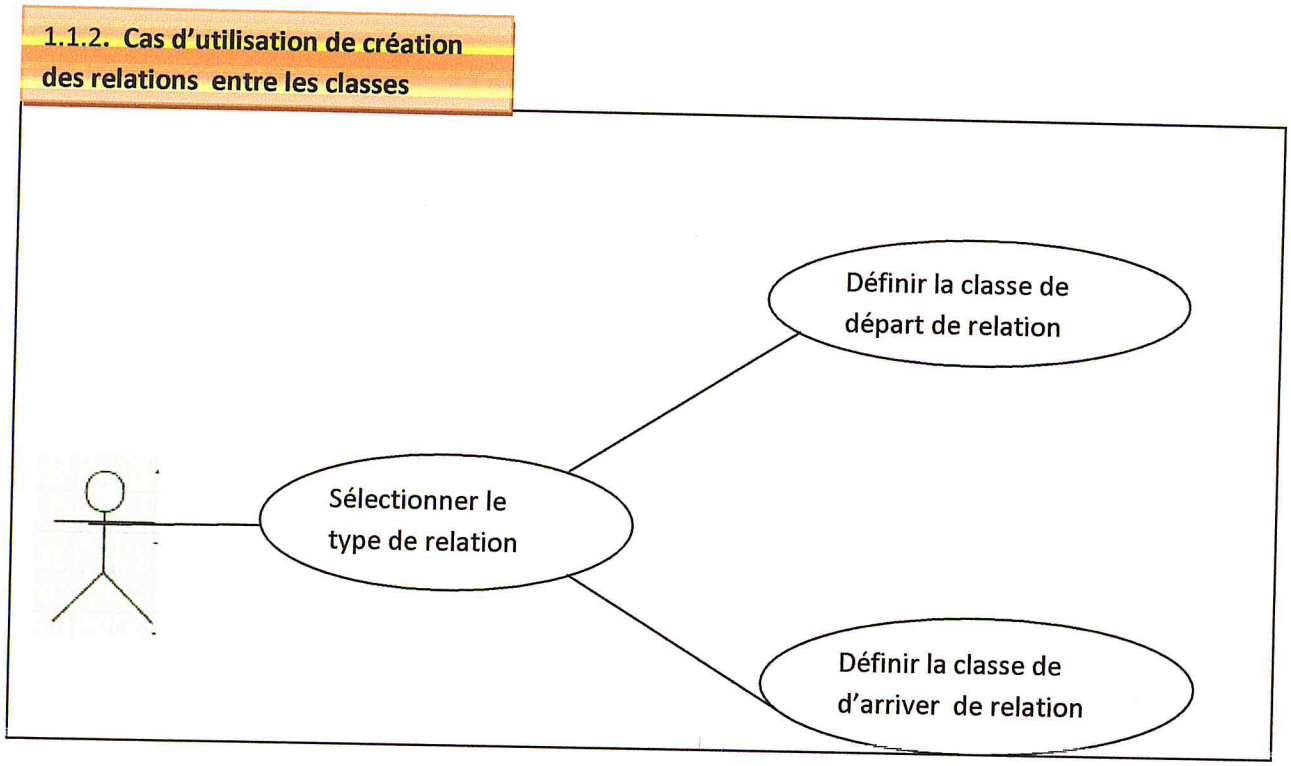


FIGURE 10: *Diagramme De Cas D'utilisation De Création Les Relation Entre Les Classes*

Et en termine par ce cas d'utilisation qui détaille une sous tache de Création du modèle source.

Donc pour réalise le modèle source on a construit une interface graphique de dessin et le la figure précédente explique le mécanisme de travail de cette interface.

2.1. Conception de L'application

L'étape de conception du svstème consiste sur la recherche de solutions qui remplissent le rôle des responsabilités définies dans l'étape d'analyse.

Nous allons convertir, dans cette phase, la description de l'analyse du système en conception qui pourrait être implémentée.

Cette phase a pour but de produire la spécification détaillée du l'application.

Les étapes de la conception du l'application sont :

- La spécification des classes de conception et les relations entre-elles.
- La production de la structure et les méthodes détaillées de ces classes (conception détaillée).

2.2. Analyse de décision

L'idée principale de conception que nous avons suivie ici est de construire une architecture à base des classes principaux et des classes déléguées qui accomplir certaine tâches du classes principales du système.

Dans le cas de notre application, il est exigé d'utiliser les patrons de conception.

Pour permettre une ***modifiabilité*** et la ***changeabilité*** du modèle source (diagramme de class), il serait plus prudent de travailler avec une interface java.

2.3. Développement de l'Application

Dans cette phase nous allons explique par des diagrammes UML le mécanisme de démarche de notre application.

2.3.1. Diagramme De Classe

Le diagramme de classes est un schéma utilisé en génie logiciel pour donner des informations générales sur les classes notre application et des relations entre elles.

Ce type de diagramme, très facile à comprendre, offre de nombreux avantages:

- il met en évidence la structure des **Classes** et leurs relations
- c'est un outil de documentation fabuleux pour
  - mettre en évidence les points clés d'une **Classe**

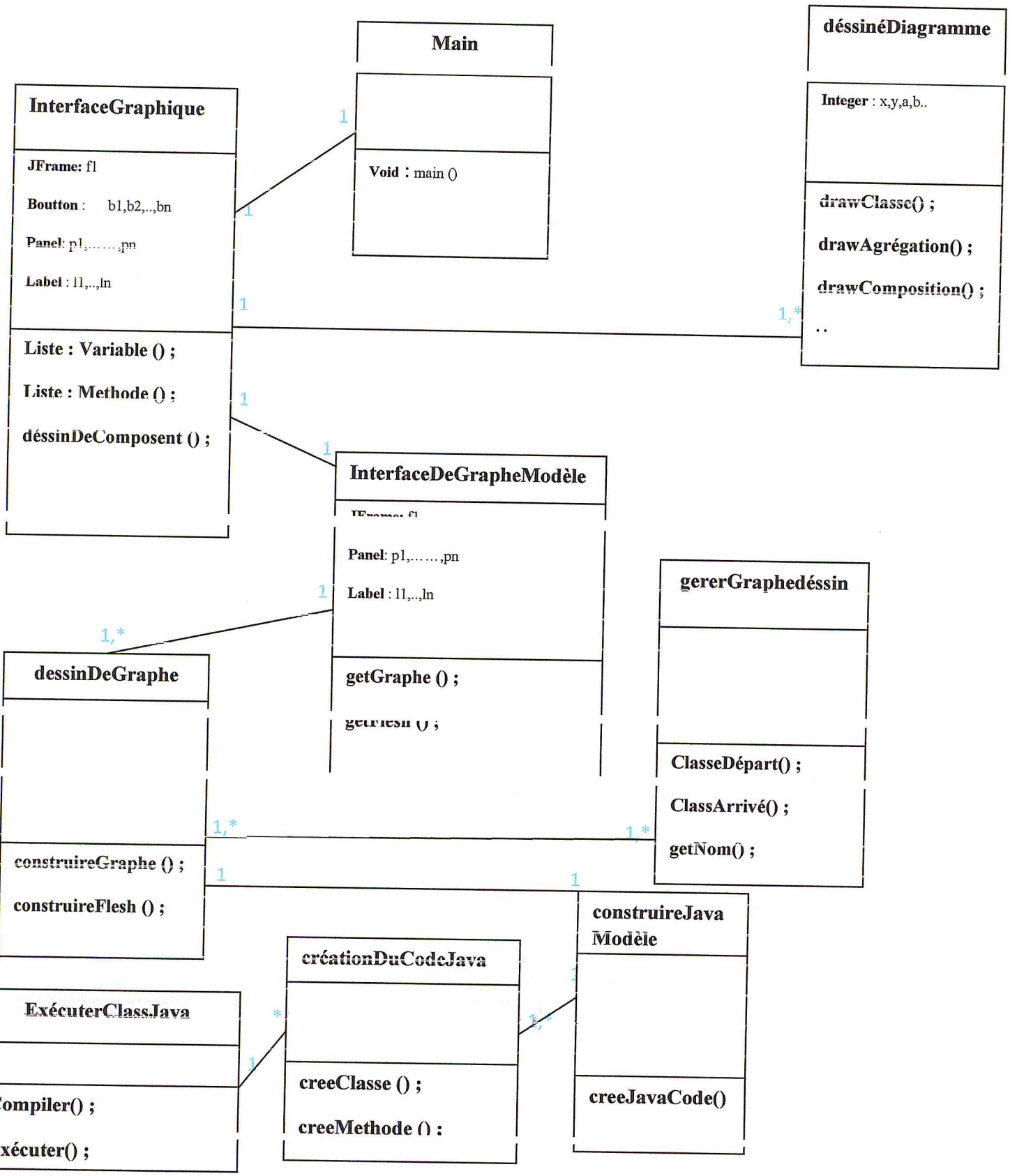


FIGURE 11 : *Diagramme significatif de l'application*

Le diagramme précédent c'est un diagramme simplifié qui permet à comprendre le déroulement de notre programme de manière très simple.

2.3.2. Diagramme de Séquence

Est un Diagramme d'interaction qui représente les objets participant à une interaction particulière et les messages qu'ils échangent organisé en séquences horaires.

Axé sur ce que fait un système et non sur la manière dont il le fait, un diagramme de séquence définit ***la Logique*** d'une instance particulière d'un ***CAS*** d'utilisation.

En général, dans un diagramme de séquence, la ***Dimension*** verticale représente les heures (de haut en bas) et ***la Dimension*** horizontale représente les différents objets.

Et notre application c'est une que suite d'interactions par des classe que en peut le résume dans ce lui :

1- l'utilisateur démarre l'application et construit le modèle source (**UML**),
2- la classe **Fen1** (l'interface graphique) lance la réalisation du modèle de graphe,
3- la classe **Fen2** (qu'elle occupe à la réalisation du modèle de graphe) passe un signale au **Fen3**,
4- La classe **Fen3** démarre la construction du code java grâce au classe **A**
5- La class **Fen3** aussi démarre la classe **B** qui permet de exécuter la classe **A** et construit la nouvelle classe qui contient un code java exécutable.
6- Et finalement l'exécution du classe permet a construit des table dans la base de données peut être utilise facilement.

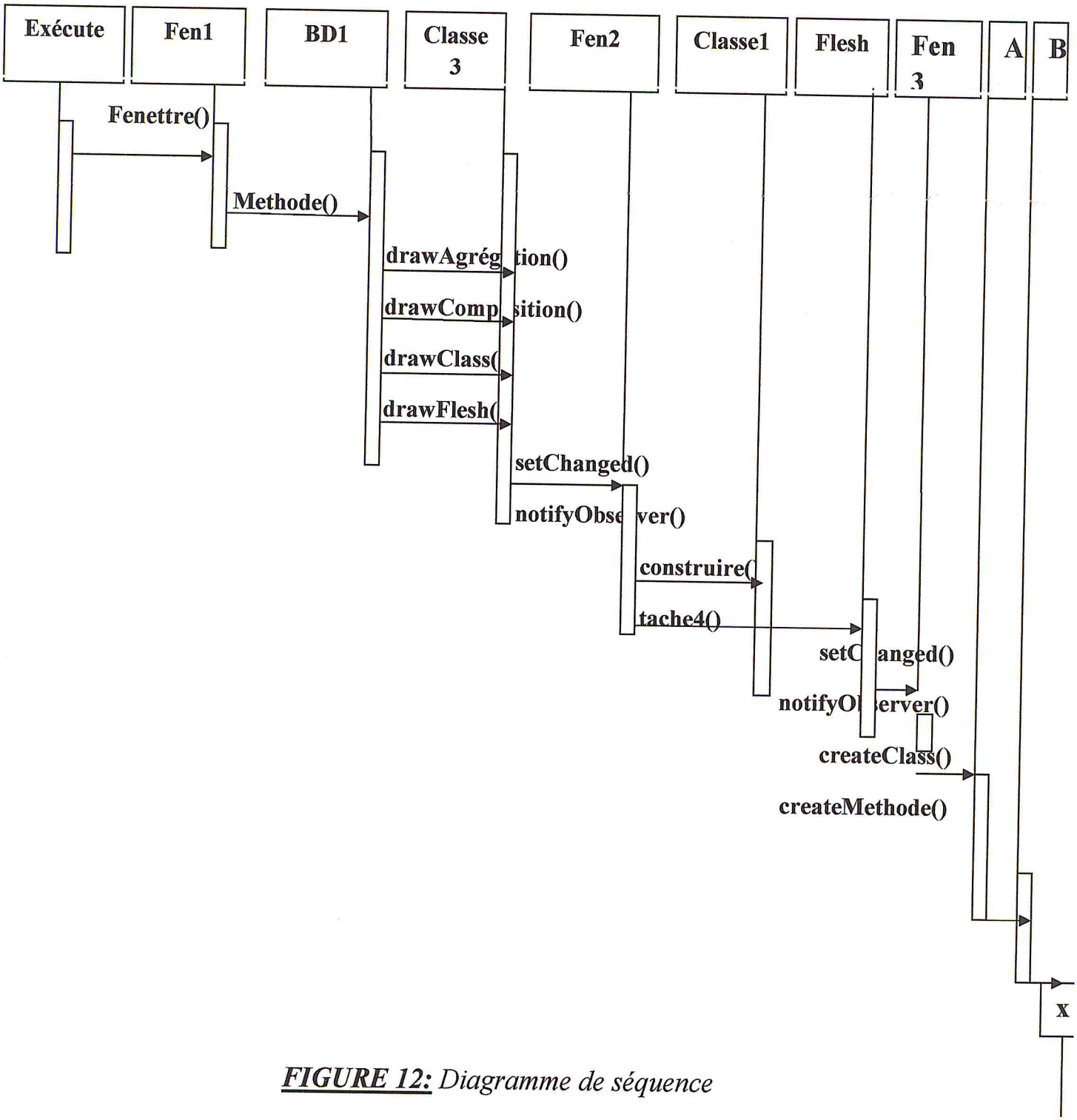


FIGURE 12: *Diagramme de séquence*

CHAPITRE 5:

Réalisation et tests

Chapitre 05:

Réalisation et Tests

1. Réalisation et Tests

1.1. Introduction

Dans ce chapitre, nous allons présenter les différents résultats des simulations que nous avons réalisées à partir d'une implémentions dont nous avons déjà présenté dans la partie conception.

Afin de valider notre application, nous commençons par présenter notre application de l'approche MDD a fin de création du le modèle java a partir du modèle source « diagramme de classe ».

1.2. Outil Informatique utilisé

Pour développer les classes utilisées dans notre approche MDD, nous avons utilisé le langage de programmation orienté objet JAVA.

Ce langage est conçu avec l'approche orientée objet, de sorte qu'en Java, tout est objet à l'exception des types primitifs (nombres entiers, nombres à virgule flottante, etc.).

Notre choix est basé sur les caractéristiques de ce langage comme, entre autres, la portabilité et l'indépendance vis-à-vis des plateformes.

De plus, Java offre une bibliothèque très riche de classes comme celles pour l'utilisation du réseau ou de l'interface utilisateur graphique. Le langage Java offre aussi le support pour l'utilisation des patrons de conceptions.

1.3. Fonctionnement

Notre application M2M (Model To Model) se décompose donc en 3 parties :

1. l'initialisation de la l'application (la réalisation de diagramme de class par le client).
2. la construction du model de graphe
3. la construction du code java.

1.4. L'initialisation de l'application

Cette partie c'est la plus délicate du projet parce que :

- Elle dessine les piliers de l'application.
- Assembler toutes les autres classes par une liaison directe (Des appels aux classes), ou indirectement par (Transfert des données).
- Un problème sur cette partie et il y aura un blocage de toute l'application.

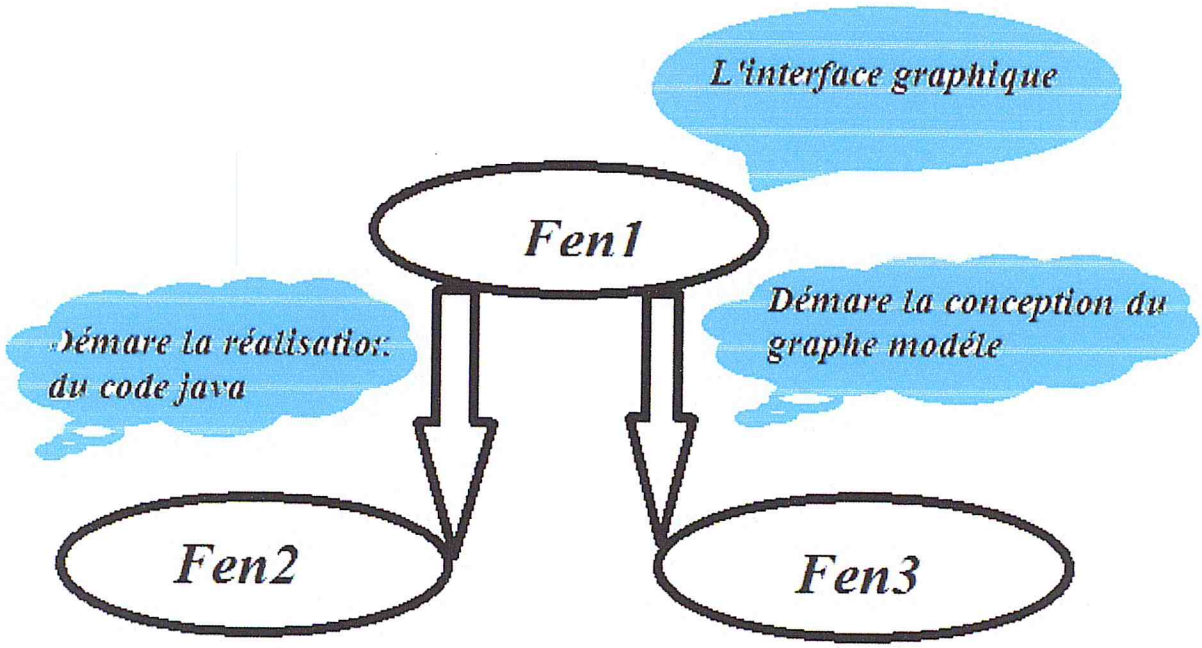


Figure 13 : L'initialisation de L'application

1.5. La Construction Du Modèle de Graphe

L'application créé un model de graphe a partir du diagramme UML qui peut être utilisable dans une autre application pour différentes raisons.

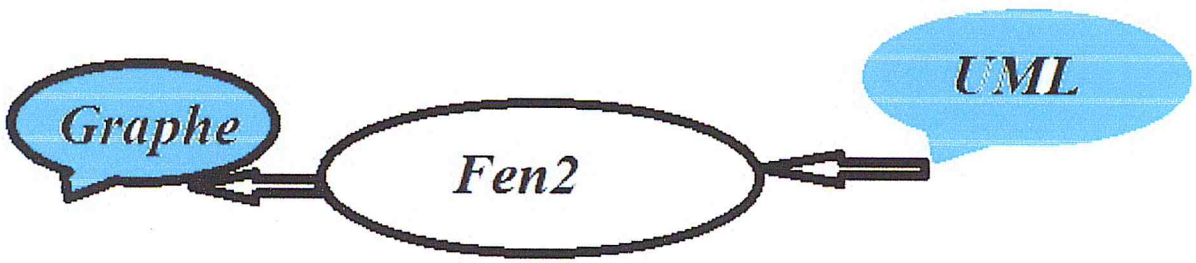


Figure 14: schéma représentant La construction du graphe

1.6. La Construction Du Code Java

Notre application construit un code java exécutable a partir du diagramme de class donc il permet le passage du schéma UML a un autre modèle qu'est le modèle java code.

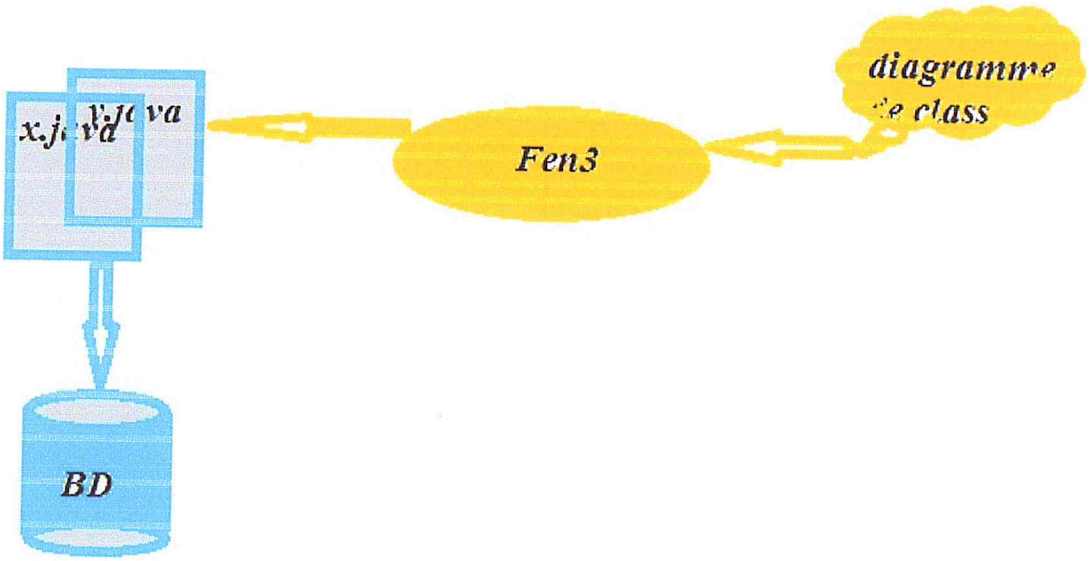


***Figure 15**: schéma représentant la construction du code java*

1.7. Schéma Général de L'application :

Notre application **M2M** est composée de 2 types de classes

- **Des classes principales** comme : Exécute, Fen1, Fen2, Fen3.
- **Des classes secondaires** d'aide comme : BD1, Class3, Variable, Class, Configue, A, B.,

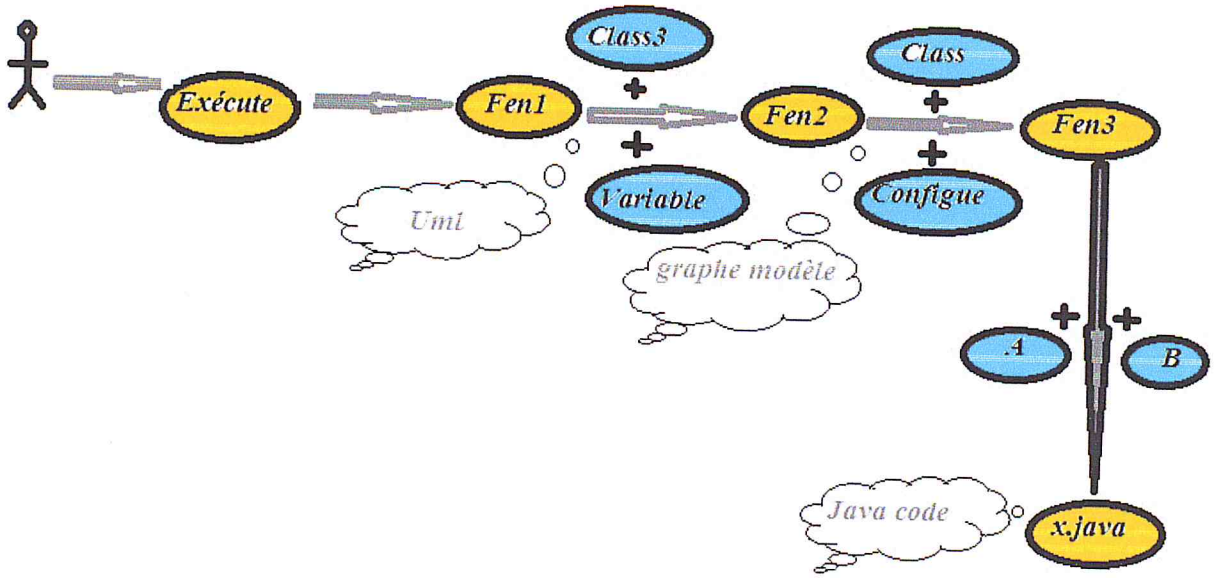


***Figure 16:** schéma générale de l'application*

Dans cette parties on démarre a expliquer l'intérêt et le rôle de chaque classe dans notre application

1.7.1. Les Classes Principales

1. La classe *Exécute*:

Cette class joue le rôle de classe principale, elle va donc permettre de démarrer l'exécution de l'application par des appels aux classes de l'interface utilisateur.

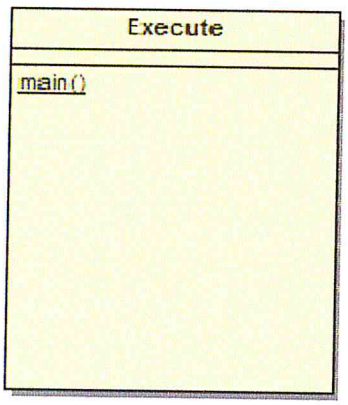


***Figure 17:** La Classe Exécuter*

2. La classe *Fen1*:

Cette classe est l'interface utilisateur qui permet de réaliser le modèle source (Diagramme de classes) qui ensuite va être transformé en un autre modèle cible (modèle de graphe ou de code java).
Cette classe est très importante dans l'application parce qu'elle :

I. Permet de construire le modèle source qui représente la matière première dans notre application,
II. Cette classe représente le point de départ de construction le modèle intermédiaire et le modèle finale (cible),
III. Une erreur a cette parties conduire a un blocage général de l'application

Principales méthodes de la classe *Fen1*:

- Les méthodes de construction des composants du modèle source comme (*tacheFlesh*, *tachAgrégation*, ….),
- Les méthodes de patrons de conception,
- Les méthodes d'action (*MouseListener* ,*ActionListener*)

a) Les Méthodes de dessin :

Dans notre application on a utilisé une interface graphique de dessin qui contient des boutons permet d'afficher les composent de modèle source grâce aux différente tache de dessin de Classe **Classe3** par exemple :

La tache agrégation : s'occupe de faire un appel a la méthode de dessiné l'agrégation dans La classe **Classe3.**
La tache Classe : s'occupe de faire un appel a la méthode de dessiné la composition dans La classe **Classe3.**

b) Les méthodes de patrons de conception :

Les méthode de conception sont des méthodes standard et bien défini permettant de d'informer toutes les classe qu'une variable ou un changement a été fait dans telle classe et la classe qui est intéressée par

ce changement va démarrer sa tache et permet aussi de minimiser les liens directs entre les classes.

c) Les Méthodes d'actions

On prend par exemple ***MouseListener*** qu'elle contient 4 méthode permet de gérer Les différents clics de souris et le déplacement sur l'interface de l'application, exemple :

public void mousePressed(MouseEvent e) {}

permet de guider la presse sur la souris

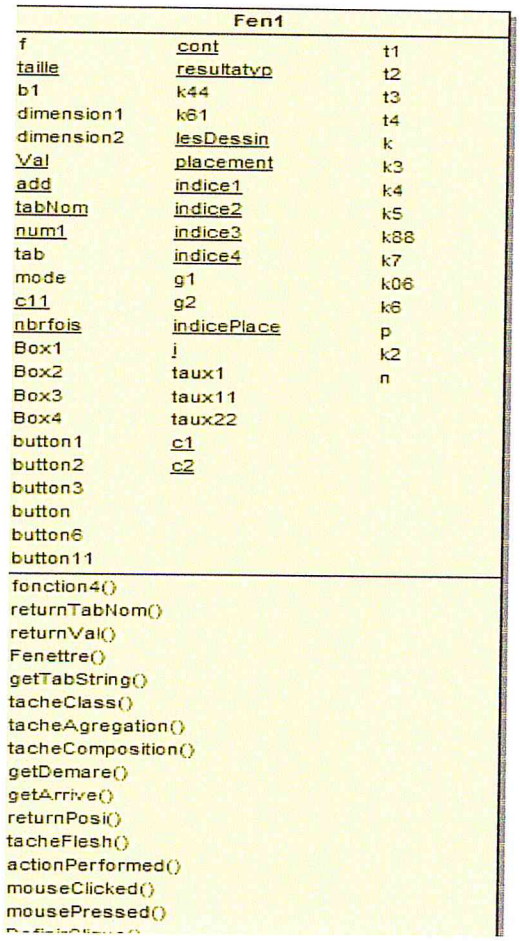


Figure 18: Classe Fen1

3. La classe *Fen2* :

Cette classe permet de:

I. réaliser à partir du modèle source le modèle de graphe,
II. construit une interface graphique pour afficher le modèle intermédiaire « modèle du graphe »,
III. transférer le modèle de graphe à la classe qui permet de réaliser le modèle java

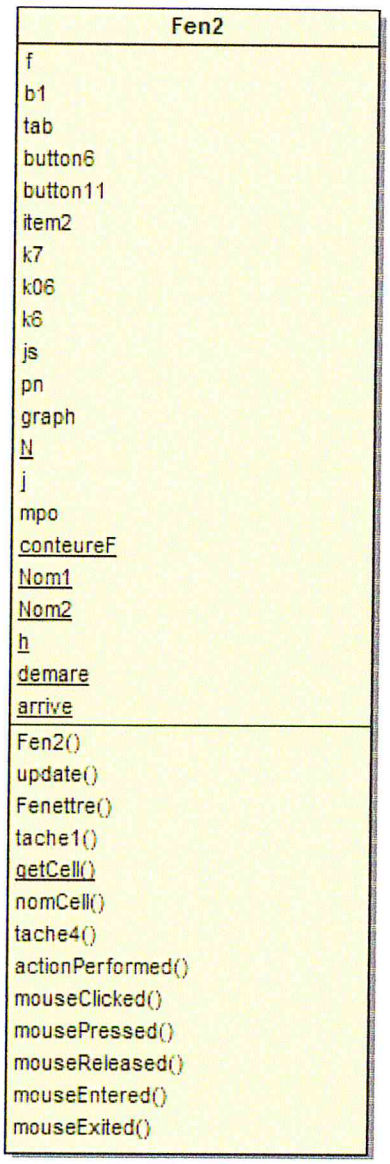


Figure 19: Classe Fen2

4. La classe *Fen3* :

Cette classe permet de démarrer la construction du code java pour le modèle source (modèle de graphe) par un appel à la classe *A* qui construit le code java.

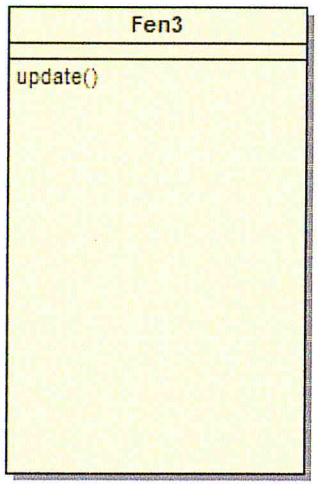


***Figure 20*: Classe Fen3**

5. La classe *A* :

Cette dernière permet de transformer une table de diagramme de classes en une classe du code java, et permet aussi de construire une table dans la base de données à partir des méthodes qu'elle comporte.

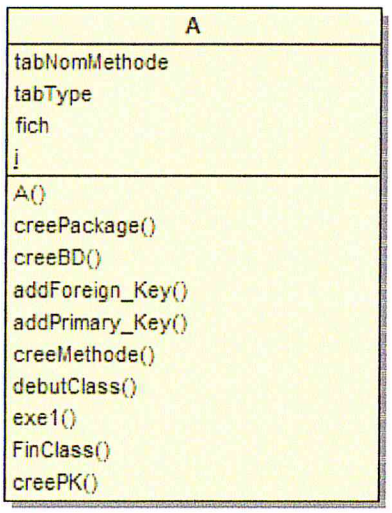


***Figure 21*: Classe A**

1.8. Les Classes Secondaires

Les classes ***Variable, B, A, BD1, Configue, Maframe, Flesh, CreeRelation*** et ***Classes3*** sont des classes d'aide aux classes principales et leur rôles sont très importants pour l'exécution de l'application.
Par exemple La classe **Classe3** c'est la classe qui permettre d'aider au dessin des différents composants du modèle source.

La Classe B :

La Classe **B** à un rôle principal d'aider la classe **A** dans sa tache et permet de :
Compiler et exécuter le code de la nouvelle classe réalisée

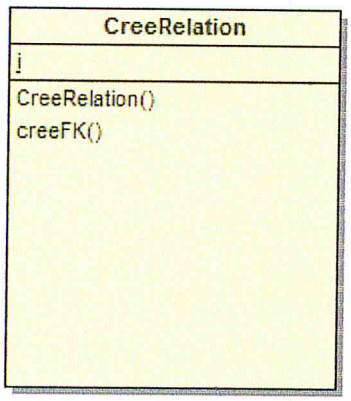


Figure 22: Classe B

La classe Variable :

Cette classe les méthodes permettant de :

I. Créer une interface graphique,
II. Récupérer les noms et les types de variable utilisé dans chaque classe construite,
III. Enregistrer et transférer les noms et les types de variable,

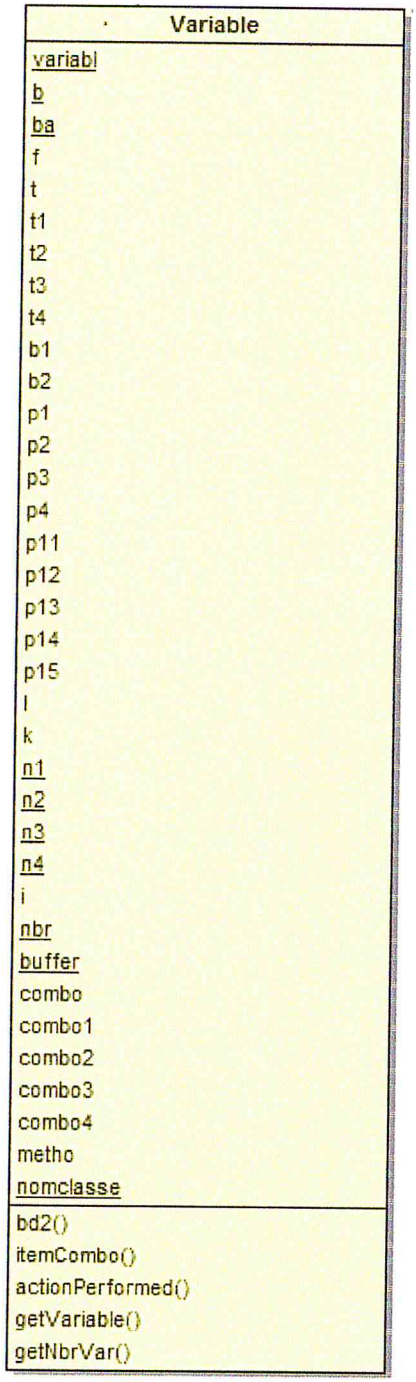


***Figure 23**: Classe Variable*

La classe BD1 :

Cette classe contient un ensemble de méthodes permettant de continuer le travail de la classe **Variable** ou de sauvegarder les différentes informations concernant les nouvelles classes construites.
Donc elle contient des méthodes pour la construction de la *base de données* ou de travailler avec le code ***SQL***.

Pour facilité la récupération et le sauvegarde de données.

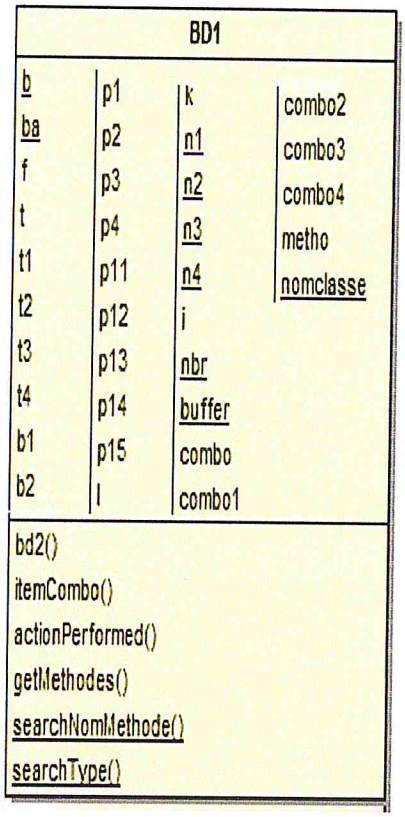


Figure 24: Classe BD1

La classe Class3 :

Elle permet de faire tous les dessins que demande la classe Fen1, grâce aux différentes méthodes de dessin qu'elle contient.

Par exemple elle permet le :

- Dessin de classe,
- Dessin des relations entre les classes d'agrégation, composition,…., et de
- redessiner chaque élément après leur déplacement.

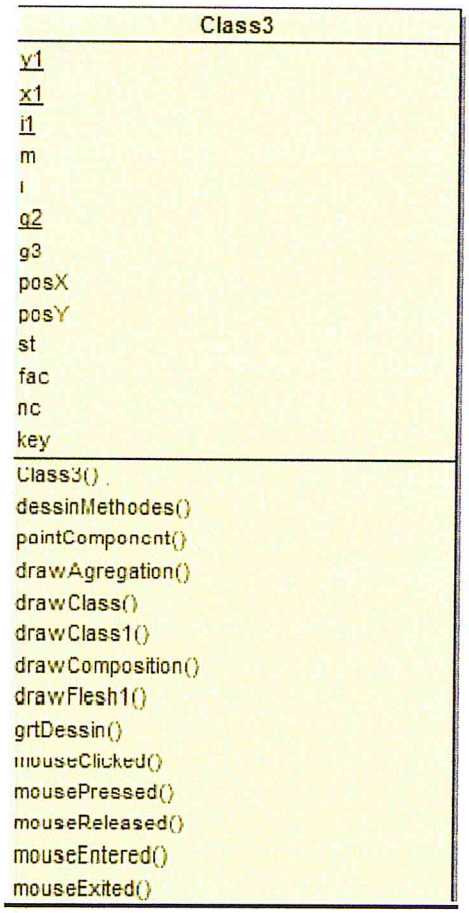


Figure 25: Classe Class3

LES TESTS

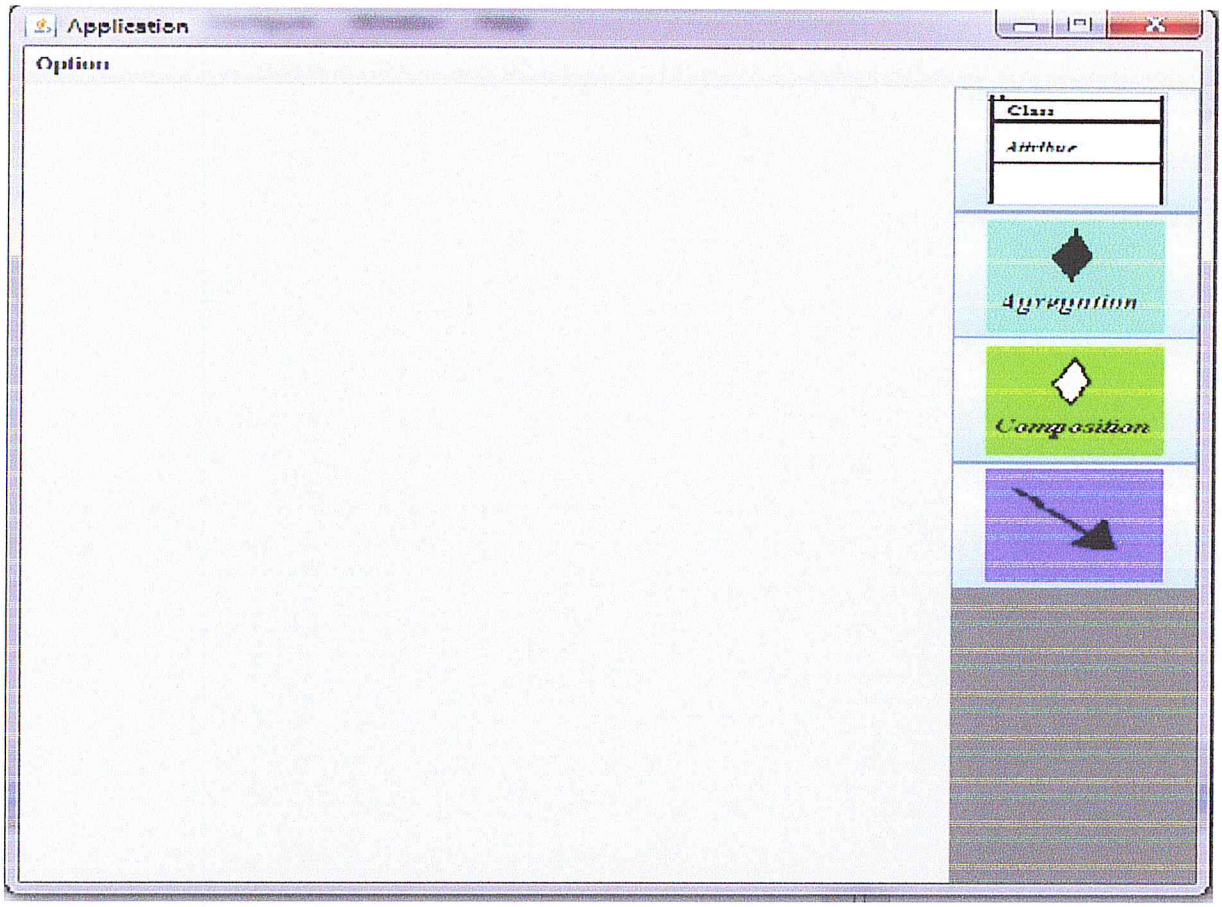


Figure 26: test1

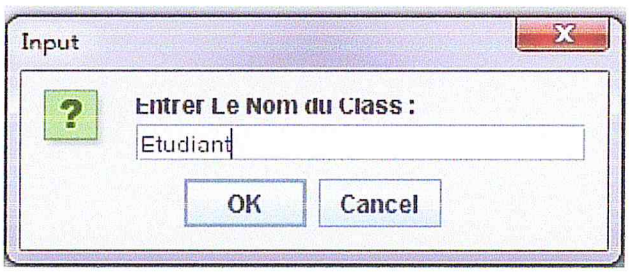


Figure 27: test2

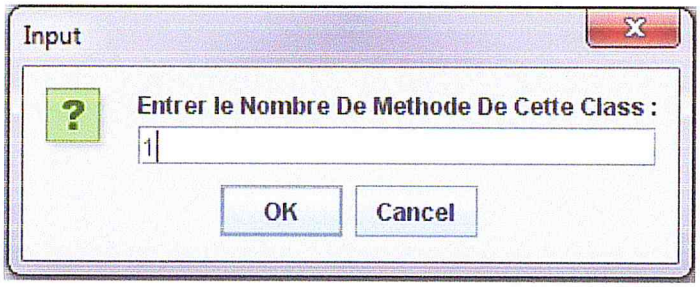


Figure 28: test3

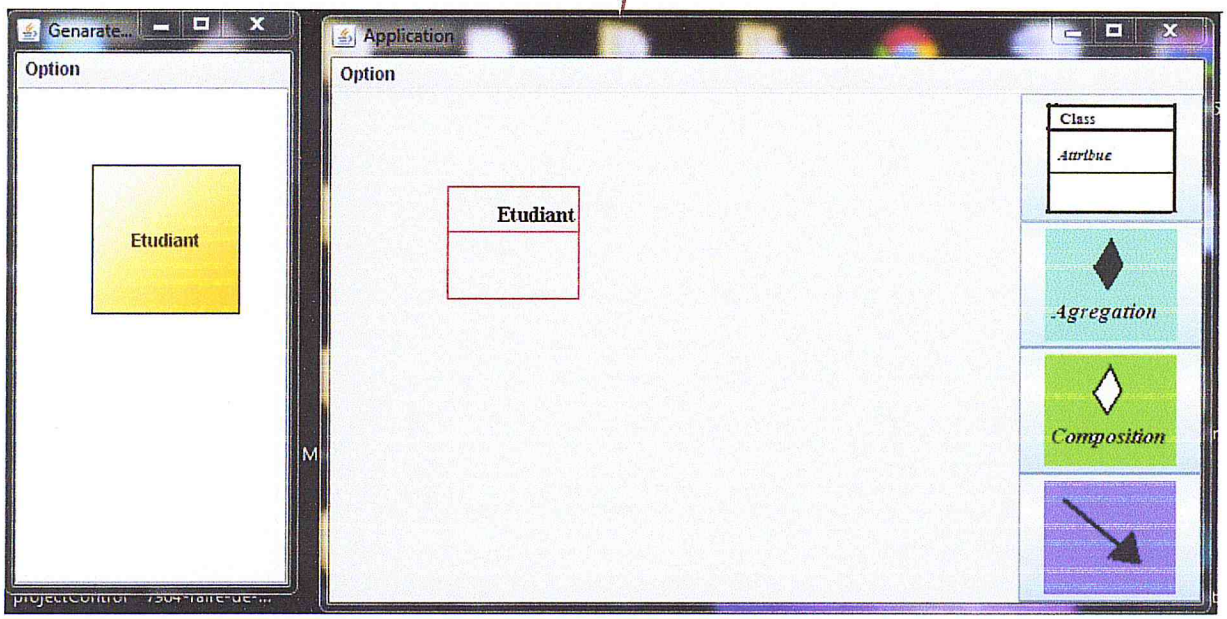


Figure 29: test4

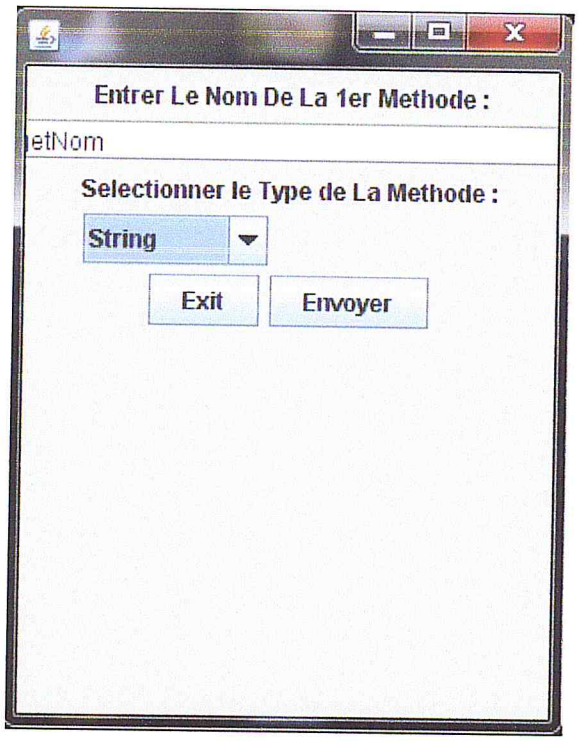


Figure 30: test5

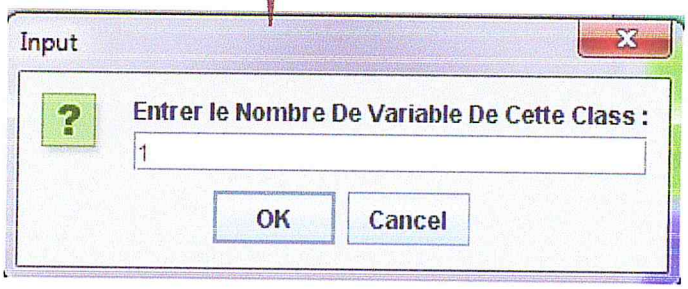
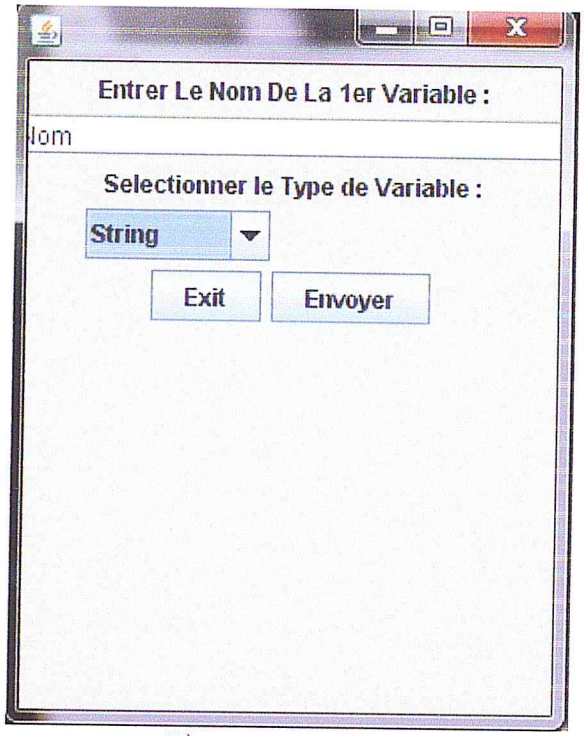


Figure 31: test6



igure 32: test7

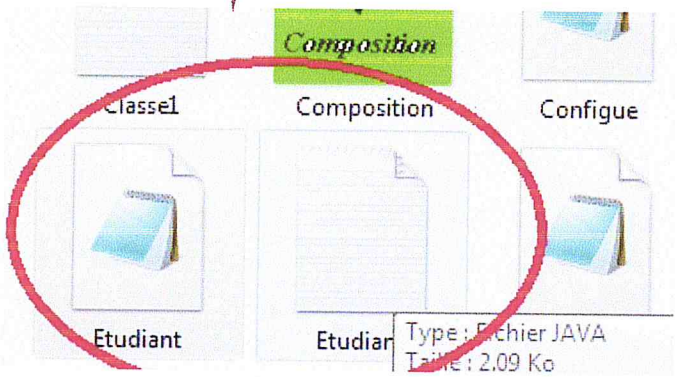


Figure 33: test8

Etudiant.java

```
import java.sql.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Etudiant{


    public Etudiant(){


    }


public void creeBD(){
    String gdbc="jdbc:odbc:mohamed2";
    String gdbc1="sun.jdbc.odbc.JdbcOdbcDriver";
    try{
        String url= "jdbc:odbc:mohamed2";
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        } catch (ClassNotFoundException e) {}
        Connection conn = DriverManager.getConnection(url,"","");
        Statement statement = conn.createStatement();
        String query1 =" CREATE TABLE Etudiant (Nom varchar(20));";
        statement.executeUpdate(query1);
    }
    catch(SQLException sqlexception)
    {
        System.out.println("SQL Exception:     "+sqlexception);
    }

}
```

Figure 34: test9

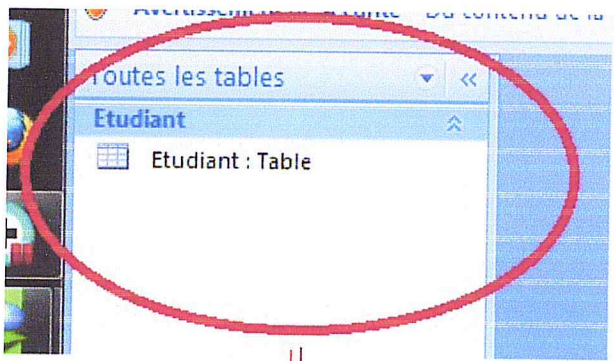


Figure 35: test10

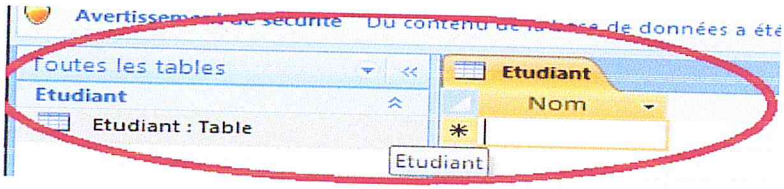


Figure 36: teste11

CHAPITRE :

Conclusion

Chapitre 06:

Conclusion

1. Conclusion

Nous avons présenté dans ce mémoire est un état de l'art sur le développement logiciel dirigé par les modèles.
Comme nous l'avons dit dans l'introduction, nous avons plus particulièrement mis l'accent dans cette étude sur les concepts, langages et outils associés à la transformation de modèles – paradigme central de l'IDM.

2. Travaux en cours et perspectives

Si des efforts importants ont été faits dans le domaine de l'IDM, cette discipline est néanmoins jeune et de nombreux axes de recherche restent à explorer :

- *Exécutabilité des modèles de processus* : l'intérêt de l'IDM est l'automatisation des transformations de modèles. Pour cela, il faut pouvoir rendre exécutables les modèles de processus qui représentent les transformations, et donc leur associer une sémantique opérationnelle. Les travaux décrits dans (Bendraou, 2007) et (Combemale, 2008) sont une première étape vers cet objectif ,

- *réutilisabilité* : pour mettre en œuvre la réutilisation de transformations considérées comme des modèles de procédés, l'un des moyens est de

définir des patrons de procédé et d'être capable d'appliquer automatiquement ces patrons pour définir ou modifier une transformation.

Nous avons élaboré une application en java pour la transformation M2M , de transformation du modèle UML en modèle de code java, en passant par un troisième modèle celui des graphes. Le modèle est important parce qu'il est très utilisé dans la conception. Le modèle de code est la finalité des transformations de modèles. Le modèle de graphes est un des plus utilisé pour exprimer des relations entre des objets de tous types.

Ce qui reste à faire à l'avenir c'est de traiter l'aspect dynamique, c'est-à-dire de pouvoir aboutir au code des méthodes à partir des autres diagrammes d'UML.

Bibliographie

Bibliographie :

[1] Johan den Haan

8 reasons why Model-Driven Development is dangerous
[En Ligne]. (25/06/2009), Electronique Source :
http://www.theenterprisearchitect.eu/archive/2009/06/25/8-reasons-why-model-driven-development-is-dangerous.

[Page consultée le 04/04/2012]

[2] Jean-Marc Jézéquel, Benoit Combemale, Didier Vojtisek.

Ingénierie Dirigée par les Modèles : des concepts a la pratique"

Editions Ellipses, 2012. ISBN : 9782729871963.

[3] ***Isabelle Thieblemont.***

« *Programmation Orientée Objet* ».

[En Ligne]. (26/03/99), Electronique Source :
http://isabelle.thieblemont.pagesperso- orange.fr/poo/poointro.htm .

[Page consultée le 04/04/2012]

[4] ***Narendra Jussien.***

« *Ingénierie Dirigé par les Modèles* ».

[En Ligne]. (2011). Eléctronique Source :
www.mines-nantes.fr/fr/Entreprise/Nos-domaines-d-expertise/Ingenierie-dirigee-par-les-Modeles .

[Page consultée Le 10/02/2012]

[5] ***Laurent Pérochon***.

« *Programme ENVOL2008* ». [En ligne].

(19/10/2008). Eléctronique Source:
https://www.projet-plume.org/files/IDM.pdf.

[Page consultée le 01/02/2012]

[6] ***Pierre-Alain Muller***, **Franck Fleurey, and Jean-Marc**.
Weaving executability into object (oriented meta-languages).
In Model Driven Engineering Languages and Systems, [pages 264–278].
[LNCS 2005].

[7] ***Laurent*** **GARNIER**.

« *MOF Query / Views / Transformations, 2007*».

[En Ligne]. (2011). Eléctronique Source :
http://www.memoireonline.com/01/12/5014/m_Strategie- de-test-au-sein-du-processus-devolution-darchitecture-de-Sodifrance22.html .

[Page consultée le 20/02/2012]

[8] ***Wikipedia***.

« *Domain-specific language* ». [En Ligne]. (2012).
Eléctronique Source :
en.wikipedia.org/wiki/Domain-specific_language .

[Page consultée le 20/02/2012]

[9] ***Eric Cariou***.

« Transformation Des Modèles ». [En Ligne]. (2012).

Eléctronique Source :
web.univ-pau.fr/~ecariou/cours/idm/cours-transfo.pdf
[Page consultée le 10/03/2012]

[10] ***Laurent Piechocki*** .

« *UML* ». [En Ligne]. (22/10/2007).

Eléctronique Source :
http://laurent-piechocki.developpez.com/uml/tutoriel/lp/cours/

[Page consultée le 10/03/2012]

[11] ***Olivier Guibert***.

« Cour UML ». [En Ligne]. (1998).

Eléctronique Source :
www.labri.fr/perso/guibert/DocumentsEnseignement/**UML**.pdf

[Page consultée le 30/03/2012]

[12] ***François Terrier***.

« *Introduction UML* ». [En Ligne]. (2005).

Eléctronique Source :
www.in2p3.fr/actions/formation/.../**Cours-UML**-LaLonde.pdf.

[Page consultée le 30/03/2012]

[13] ***Wikipedia***.

« ATLAS Transformation Language ». [En Ligne]. (2007).

Eléctronique Source :
fr.wikipedia.org/wiki/ATLAS_Transformation_Language.

[Page consultée le 30/03/2012]

[14] ***ECLIPSE***.

Eléctronique Source :
"ATL". [En Ligne]. (2007) www.eclipse.org/atl/

[Page consultée le 25/04/2012]

[15] ***Mathieu G***.

« Introduction au Design pattern ».

[En Ligne]. (22/04/2007)

Electronique Source :
http://design-patterns.fr/introduction-aux-design-patterns.

[Page consultée le 5/04/2012]

[16] **« Les Patrons ».**

[En Ligne]. (22/04/2007)

Eléctronique Source :
http://sourcemaking.com/design_patterns.

[Page consultée le 5/04/2012]

[17] ***Eric Sigward.***

« Les Graphes ». [En Ligne]. (22/04/2007)

Eléctronique Source :

http://www.acnancymetz.fr/enseign/maths/m2002/institut/ipr/graphes/htm-l/definition.htm .

[Page consultée le 5/04/2012].

[18] The Object Management Group (OMG),

Eléctronique Source :
http://www.omg.org.
[Page consultée le 15/03/2012].

[19] Les Graphes
Eléctronique Source :
hidouci.esi.dz/mcp/2_**Graphes**.pdf.
[Page consultée le 20/04/2012].

[21] **Bézivin J, Blanc X.**
Promesses et Interrogations de l'approche MDA.
Journal DéveloppeuRéférence,
[En Ligne]. Septembre 2002,
http://www.devreference.net/,
[Page consultée le 5/04/2012].

[22] **Bézivin J.**
La transformation de modèles. INRIA-ATLAS & Université de Nantes,
Ecole d'Eté d'Informatique CEA EDF INRIA, cours #6, [2003],

[23] **Bézivin J.**
Sur les principes de base de l'ingénierie des modèles.
RTSI-L'Objet, [Page145-157], 2004.

[24] **Yvan Radenac.**
Mini-mémoire: Les langages orientés objets et leurs implémentations libres [10/05/2002]

[25] OptimalJ - Model-driven development for Java,
Eléctronique Source:
http://www.compuware.com/products/optimalj/, 2003
[Page consultée le 1/04/2012].

[26] **Scott W, Ambler A.**
Model-Driven Development. (23 February 2007).
Eléctronique Source:
http://www.agilemodeling.com.
[Page consultée le 2/04/2012].

[27] **Shaw, M.**
"Patterns for Software Architectures", In: Coplien,
Pattern Languages of Program Design. Reading: Addison-Wesley, pp. 453-462, 1995.

[28] **Gamma E., Helm R.**
Johnson R. E. et Vlissides J., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley, Reading, Massachusetts (USA), 1994.

[29] **Buschmann F, Meunier R, Rohnert H ET Sommerlad P.**
"Pattern-Oriented Software Architecture - A System of Patterns", Wiley, John Sons, Incorporated, 1996.

[30] **Stahl T, Volter M.**
Model-Driven Software Development. Wiley. 2005.
Laurent Pérochon, programme ENVOL2008, Annecy 19 au 24 octobre 2008, pages 428.

[31] **Jean Bézivin, Mireille Blay, Mokrane Bouzeghoub, Jacky Estublier, Jean Marie Favre.**
IDM Ingénierie Modèle CNRS.pdf
Eléctronique Source :
idm.imag.fr
[Page consultée le 2/01/2012].

[32] **Franck Fleurey, Jim Steel, and Benoit Baudry.**
Validation in Model-Driven Engineering: Testing Model Transformations., First International Workshop on Model, Design and Validation., [pages 29–40]. IEEE Computer Society, 2004.

[33] TOPCASED-WP5, staff, Guide méthodologique pour les transformations de modèles.Rapport de recherche n° 8, Novembre 2008, IRIT/MACAO

[34] **Burmester S, Giese H, Niere J, Tichy M, Wadsack J, Wagner R, Wendehals L, et Zundorf A.**
Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 6 (3), page 203-218, 2004.

[35] Xactium-website, XMF-Mosaic.

Eléctronique Source :

http://www.xactium.com

[Page consultée le 19/05/2012].

[36] Triskell, Kermeta, IRISA, Rennes,

Eléctronique Source : 2005

http://www.irisa.fr/triskell, http://www.kermeta.org
[Page consultée le 15/05/2012].

[37] **Jézéquel J, Fleurey F, Drey Z, Muller P, Pantel M, Maurel C.**
Kermeta . In *IDM'05, Paris Premières Journées sur l'Ingénierie Dirigée par les Modèles*, Posters & Démos, Paris, 30 Juin- 01 Juillet, 2005.

[38] **Eclipse-EMF**,
Eléctronique Source :

http://www.eclipsetotale.com/index.html?keywords=emf;
http://www.eclipse.org/modeling/emf/
[Page consultée le 13/05/2012].